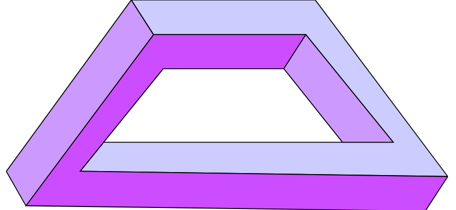


Business Rules For Object-Oriented Projects

By the Staff at Management Strategies

Application Architecture Is Fundamental

Managers and developers adopt object-oriented techniques, not for the benefits of the new technology, but for protection against the weaknesses of the old technology. They are usually seeking refuge from the overwhelming complexity and rigidity of first-generation client server technology. Whether they actually find a safe haven depends very much on whether they master the new architecture.

The experience we relate in this paper comes from direct participation and first-hand observation of O-O projects at several large insurance companies, two telecommunication giants, and a leading multi-national financial services firm.

One underlying theme in these projects was the belief that the O-O paradigm relieves developers of responsibilities of sound software engineering because of the magical, ineffable properties of classes and objects. Those who believe this are sorely mistaken.

Quite the contrary, the need for application architecture is as great or greater in O-O projects. As we have already stated, these are large and complex projects. The object paradigm is nothing if not a process of abstraction. Poor abstraction leads to poor results just as in "conventional" programming. Furthermore, the O-O paradigm, by itself, says nothing about two critical dimensions of the application architecture: workflow and the dynamics of application design. Unfortunately, in many cases, the application architecture often evolves as a second cousin of the technical architecture.

None of the O-O projects we observed was successful by any traditional measures: they consumed tens of millions of dollars of resources, produced substandard products, and were significantly late. Nevertheless, we, and the managers who worked on them, learned some valuable lessons about serving the needs of the business community. The primary lesson is that a comprehensive application architecture is necessary if O-O projects — indeed any projects — can succeed.

Application architecture defines the way that the objects in a software system behave from the viewpoint of the business. Once we have defined the

fundamental modes of behavior and interaction, development can proceed in an orderly fashion.

Concerns about architecture typically begin and end with the definition of technical “subsystems." O-O developers usually only discuss these concerns in a treatment of the construction of class hierarchy and encapsulation. The flaw is that these concerns go no further than the boundaries of the technical aspects of the discipline.

Recent work by Gamma[1], et al., and Pree[2] indicate the direction that we must pursue if we are to adequately reap benefits of O-O technology by revamping how we manage O-O projects. Simply stated, this work explores the role of patterns of assembly of O-O design primitives (objects, classes, properties, and methods). These software design patterns allow engineers to apply standard solutions to common design problems.

While this work is valuable, it is still too technically oriented to be of much use to a beleaguered business manager, harried by time and budget constraints, and an impatient user community, who can’t appreciate the complex engineering issues.

No matter how fine a job we do building the engine beneath the hood of the O-O car, practical O-O managers must still match up the business requirements with the technical characteristics of the delivered software product. In a sense, this is just an extension of the age-old challenge that all system builders face.

Commercial software developments generally have relatively tight time-frames and budgetary constraints. For that reason, attaining, and maintaining focus on business objectives is one of the most critical factors for success. And this is where many O-O projects encounter development difficulties.

Business People Don't Care About OOP

We developers like to think that object-oriented programming and design is "natural," "intuitive," and all-around easier for business users to understand. All too often, once we make this claim, we immediately hide its kernel of truth in a fog of O-O jargon.

Recently, several managers in one company complained to us that the object-oriented technologists had nearly derailed a major re-engineering project because they had introduced object-oriented terminology that most of the business people took to be a foreign language. This is hardly the way to win any sort of significant commitment from the business community.

Although, as software developers, we cannot completely adopt the language and style of the business community, we can adapt our discourse to engender greater clarity with our user constituents. We must find a medium within which it is possible to discuss the pertinent requirements of the business community, while

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[2] Wolfgang Pree, Design Patterns for Object-Oriented Software Development, Addison-Wesley, 1994.

still conforming to the highly technical environment of object-oriented program development.

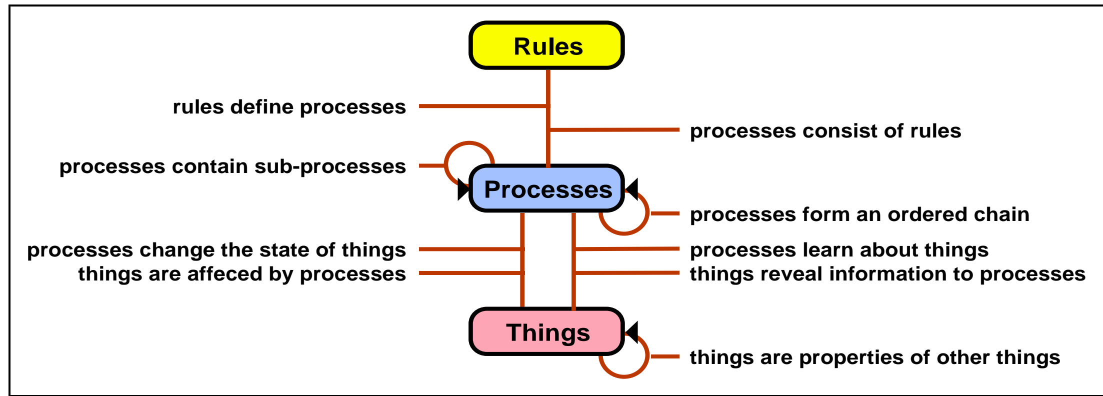
An object-oriented application architecture in general, and business rules in particular, constitute that happy medium. Before we can build the architecture, however, we must first establish a ground floor of communication and understanding.

Business ABCs

It is the responsibility of IS managers to learn the language of business and not the other way around. When talking about business activity, business people talk about "***rules***", "***processes***", and "***things***". These are the ABCs of business. Almost everything that can happen in business falls under one of these three basic categories.

IS managers should communicate with their business peers in nomenclature that is useful for requirements gathering and project management. These terms can even also translate into program development since they map directly into the more technical realm of object-oriented design. Object-oriented design, in turn, leads to applications that are easier to build, more flexible, and less costly to maintain.

By using *rules*, *processes*, and *things* at the design stage, IS managers can better communicate with their user constituents, while improving the quality and responsiveness of their development services.



Building Bridges

Businesses succeed when communication succeeds. Good communication builds bridges between employees, as well as the customers they serve. In successful organizations, these bridges extend from the last hired hand to the board of directors. These bridges rest upon the twin foundations of simplicity and clarity. Good business communication is intelligible to everybody in the company who needs to understand it. Often, perhaps usually, everybody in the company can understand it--period.

If Information Systems developers are to succeed they must build communications bridges to their business users. There is no hope that these

business users will learn the language of technology. (Why should they, when technologists themselves can't seem to agree on terms?) Instead, developers must learn to express their work in the language of everyday business.

Business people have no trouble describing everyday business activity in everyday English. The mission of software developers is to automate business activity. Why shouldn't developers be able to discuss their own activity in these same everyday terms? In fact, why can't developers bridge from everyday business English right down to the most subtle technical nuance? We think developers can, and if they are to succeed, must do just this.

Putting Rules, Processes, and Things to Use

The astute reader has no doubt noticed that these concepts can map into the technical arcana of information engineering, object-oriented analysis, artificial intelligence, etc. From our perspective, we care about two basic things:

Our end-user clients think in terms of rules, processes, and things.

We can model systems in terms of rules, processes, and things.

Surely, if our intent is to communicate, we should talk to end-users about rules, processes, and things. Later, if we wish, we can turn them into production rules, constraints, functions, procedures, triggers, methods, objects, classes, entities, attributes, relationships, variables, parameters, tables, columns, domains, slots, frames, properties, data flows, state-changes, interfaces, messages, function calls, lists, sets, bags, or whatever we want to call them in the privacy of our cubicles.

The principal and underlying assumptions are that business processes represent the methods of objects, on one hand, and business rules can represent all business processes, on the other. The designer can formulate a rather complete picture of the application requirements by asking the business user about these three things. Consider the following three statements:

If the order takes the customer over the credit limit, don't post it.

Enter the order, check the credit, then post the order.

Every order has a customer, line items, and a total.

First, note that these three statements are about rules, processes, and things. They are about business--not technology. There is nothing in any of them that tells you whether the implementation is object-oriented. In fact, there is no hint that these operations are computer-automated at all!

The first statement is clearly a business rule. A rule describes project information at its most detailed level. A single rule in the basic “WHEN...IF...THEN...” form translates into no more than a handful of lines of code.

The second statement is clearly a description of business processes. It is no coincidence that each of these processes has the scope of a method of an object. Methods are "natural" in the same way that objects are "natural."

The third statement describes the object model itself. Specifically it defines an *Order* class and three of its properties: *Customer*, *LineItems*, and *Total*.

That these three statements interrelate logically, hardly needs mentioning. Their logical interrelationship corresponds to a technical interrelationship on the O-O paradigm. The three processes are methods of the *Order* class. One of these methods, *checkCredit*, executes (among other things) the rule in the first statement.

Business people, even those who are quite naive about computers and object technology find it relatively easy to describe their business processes in straightforward English language. Technologists must translate from one domain to the other: that is from the domain of the business person to the relatively more constrained and precise world of technology. A major advantage of the rules/processes/things model is that it puts the O-O developer in a better position to match up the implemented system to the actual business concerns of the ultimate system users.

In one major insurance company we worked with, the O-O development community completely lost sight of the core functionality of the business. Engineering frameworks that were extremely flexible but provided little or no problem-solving benefit to the user community devoured tremendous amounts of time, money, and intellectual resources. Only by significantly paring down end-user functionality were the developers eventually able to deliver a workable set of functions.

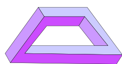
Business Requirements for the Application Architecture

The application architecture exists to satisfy business requirements. Today, changing business requirements have a profound effect on the application architecture. Business process re-engineering requires an architecture that is more integrated, yet more flexible, than the old "one machine/one application/one function" mainframe architectures. Here are some of the main business issues that any application architecture must address:

- **The architecture must fit the requirements of the application it supports.** If this sounds like a tautology to you, ask yourself these questions. First, are you considering a move to a distributed application architecture? Second, if you are, what business requirement is driving this move? If only technical requirements drive the switch, it is probably a mistake.

- **The architecture must fit the capabilities of the development organization.** If it doesn't, your project is almost certain to fail. It is very hard--perhaps impossible--to move a whole organization from COBOL to C++ or Smalltalk or Forte.

- **The architecture must minimize risk.** Half or more of all new software projects fail. The reason for this has to do with managing risk--especially technical risk. New software technology is coming much faster than IS organizations can learn. Risk management means introducing new technology when and only when business



requirements call for it and your development organization knows how to use it.

- **The architecture must be adaptable.** Business requirements change. Today they change rapidly. When they change, software systems must change with them. Often, legacy systems simply can't make the change. When this happens, it is time to rebuild. When rebuilding systems, there is no greater mistake than to repeat the error of building inflexible systems. The new systems must have an architecture that can adapt to the next wave of changes.

- **The architecture must be reusable.** We usually see this as the first requirement of an object environment, because we think of re-usable objects as being the foundation of an adaptable architecture. Without re-usable objects, changes of any sort to large systems — for process re-engineering, for system expansion, or for routine maintenance — become difficult or impossible.

Components of the Application Architecture

Today, most developers accept O-O design and technology as the standard ways to achieve reusability. In general, this perception is correct. However, O-O design and O-O technology by themselves, cannot meet the dynamic and changing needs of business.

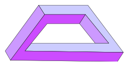
To see why this is so, just think about an archetypal O-O application — say a graphic user interface or a drawing program. This type of program provides its user with a rich array of functionality to put an object on the screen, move it around, change its properties, and, when finished, to destroy it. But what if many users had to work on the same screen (or, more realistically, the same drawing) at the same time? What if you wish to change the basic behavior of the application mid-stream, without waiting for the next release from Microsoft or Lotus?

The basic O-O model of class, property, method, and instance is not enough. Now you must clearly define basic object behavior in such a way that you can easily change its control sequence without reprogramming. You must also build the control sequence into the run-time model.

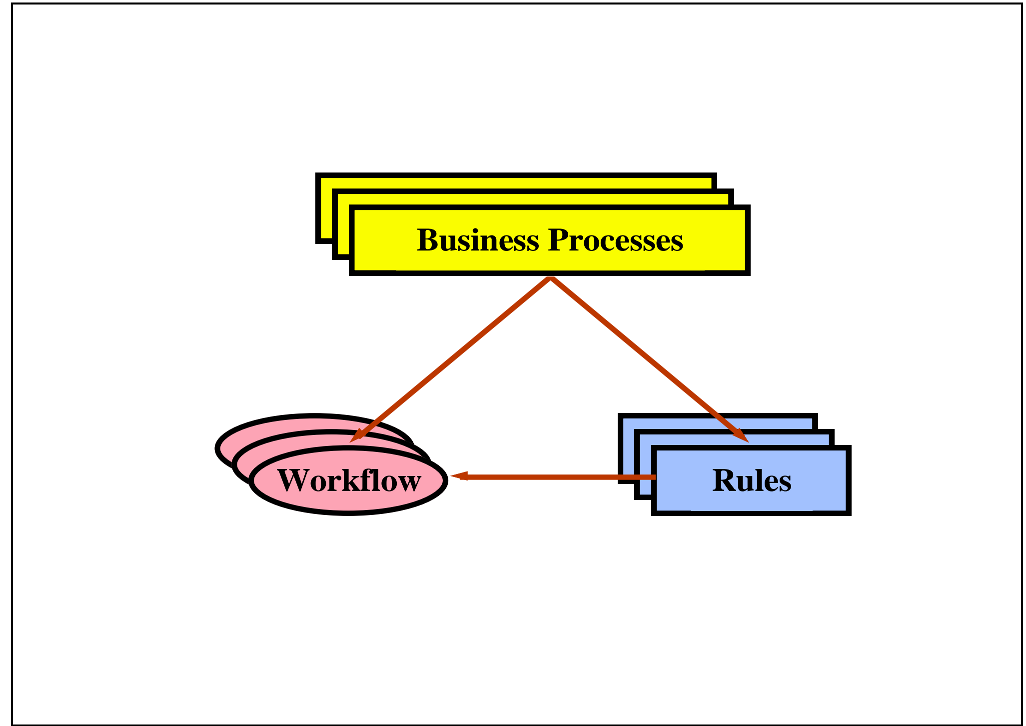
To meet these expanded needs we must introduce rules as a separate model component to embody control sequence outside of hard-coded methods. We must also introduce a notion of workflow to keep track of the twists and turns of each particular control sequence.

In this operation, one business process activates a rule which evaluates the state of an object, then, in turn, activates another business process. Business processes, like any other process, are the implementation of the methods of objects. However, they are explicitly different from purely technical processes. Put simply, business processes are methods that business people care about.

The end result of this kind of thinking is a run-time architecture based on rules, business processes, and workflow (and inference objects) that defines only the



component of applications of interest to business managers. In its essence, this run-time architecture is extremely simple:



Note that this highly simplified sketch is independent of every aspect of the technical implementation. There is no indication of programming language, hardware platform, operating system, network, messaging system, or any other technology. This piece of the overall architecture speaks to business issues only.

However, this design is the cornerstone of an adaptable technical architecture. Once we have clearly articulated business processes, rules, and workflow, it is possible to create a technical architecture capable of changing to meet local and temporal business needs.

A Definition of Rules

Rules include all those things you and we would ordinarily think of as rules in the business sense; "Payday is Friday" or "Ship orders within 24 hours of their receipt."

More importantly, rules are part and parcel of the business person's everyday vocabulary. While we technologists may regard them as a collection of requirements (and indeed they are), business people generally see them as something rather more important than mere requirements. Often these rules have enormous consequences for profit and loss, good customer service or bad, high-quality goods or unacceptable production.

The secret to success is to tie the architecture to these rules. When these rules change, make sure the application system can change just as quickly, and above in an economic manner.

In the following sections, we discuss the design of an advanced technical implementation of this architecture.

Design Patterns Business Rules

So far, O-O technology has fallen short on its promise to simplify the application development process. Too many large O-O projects fail. Gamma, et al. and Pree explore the role of design patterns in the assembly of O-O applications. These software design patterns allow engineers to apply standard solutions to common design patterns.

We can use this approach as a starting point to understand rules. Let us begin by defining a design pattern as a form of a software template. If you have ever used a template written by someone else to write a routine or simply cannibalized some of your own code to suit a new purpose, you have the idea. Even if you aren't a programmer, the ordinary meaning of a template applies. The basic idea is that you start your work with something fixed and constant, then make changes as you go.

Pree goes one step further than Gamma in suggesting that we can distill design patterns into "meta-patterns." Meta-patterns are fundamental building blocks which make up any of a vast universe of problem-specific design patterns. Pree identifies ten such meta-patterns.

A design pattern is a set of relationships among a group of objects that is not unique, but occurs over and over again in different application designs or even among different groups of objects within a single application. When a pattern occurs, some parts of an object will be fixed or constant, while other parts will vary. Pree calls the variable parts of the pattern "hot spots." He illustrates the concept as follows:

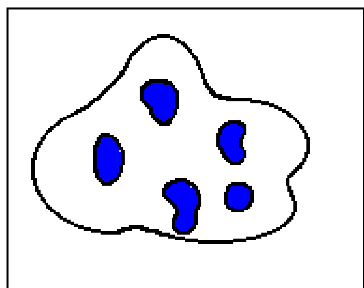


Figure 0-1

In this illustration the large shape represents the totality of a group of interrelated objects (which Pree calls a "framework"). The blue shapes are hot spots. The art of using patterns for O-O design, then, is to separate the variable hot spots from the fixed part of the pattern. The hot spots become separate, stand alone methods, either in the same object as the fixed code, or in an entirely new object. The designer can then vary the pattern by varying the implementation of the hot spot code. These variations can include inheritance, code generation, or parameterization.

Three Views of Business Rules

There are three primary views of business rules of interest to object-oriented development managers.

These are:

- Run-time Abstraction view
- Development Environment view
- Business Person's view

The Run-time Abstraction view encompasses the execution environment. Unfortunately most developers barely look beyond this horizon. The Development Environment view covers the practical components necessary to manage and control the rules environment. The Business Person view is the user interface to the rules environment, and it is the most important of the three major perspectives. Even if the more technical aspects of the rules environment were to be outstanding, lack of a good user interface would almost certainly lead to the occurrence of many of the problems we have observed in the field.

Let us examine each of these perspectives in turn, starting first with the run-time abstraction view where many of us technically oriented practitioners feel most comfortable, and then moving upstream through the development environment to finally deal with the realm of the Business Person.

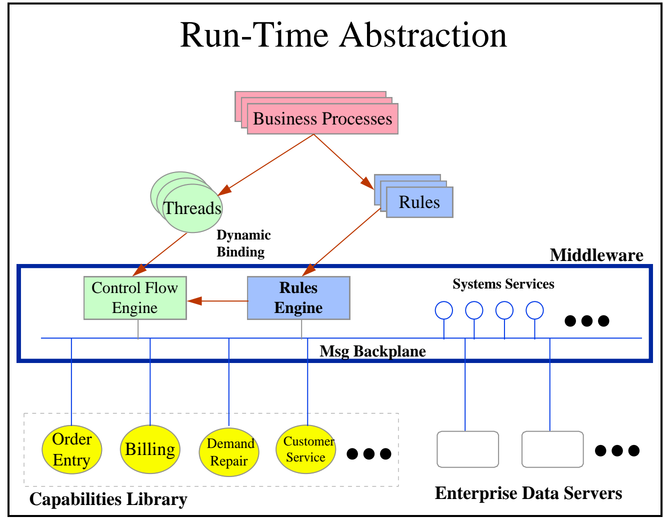
Run-time Abstraction View

Business processes, as we often say, are the heart and soul of application architecture.

What we present below is a summary of a model architecture we helped develop at the GTE Telephone Operations Center. While no single company has implemented an architecture with all the features of this one, we have observed elements of this architecture working quite well in several major Fortune 500 companies

The two central elements of this framework are a Business Rules Engine and a Workflow Engine. Our colleagues and we have reported some of the early work and the conceptual underpinnings of this framework in a separate paper[3].

[3] Mac Clark, William Frank, Mary Johndrew, "Business Rules Management and Execution," Third International Conference on System Integration, Sao Paulo, 1994.



Business Processes	These are the processes by which the business achieves its objectives and runs orderly operations: e.g. order entry, checking inventory, checking credit, and posting an order.
Workflow	Workflow is very much like the concept of a thread in multi-processing systems: It is an independent sequence of steps, a particular route through all the possible twists and turns of a particular business process.
Workflow Engine	Handles the control, scheduling, and management of workflow, procedures, process steps, and transactions.
Rules Engine	This layer accesses rules from a rules repository, translating rules into executable statements, and executing the rules in a particular environment.
Capabilities Library	These capabilities correspond to the library of reusable application-level objects.
Middleware Layer	Responsible for the guaranteed delivery of message traffic across a network of cooperating processes.
Message Backplane	This layer allows separately developed services and platforms to support application execution
Data Servers	These servers establish the data resources available to application systems
System Services	Often-used routines, e.g. date, time, security, that

	do not change; but usually require more specialized engineering than the capability library.

Development View

A critical element of the application architecture is how developers map the rules of the business onto technology components: that is, classes, objects, methods, subsystems, and systems. Here we illustrate a rules management framework that we developed in conjunction with representatives from GTE Telephone Operations, GTE Data Services, and GTE Laboratories in an intensive workshop.

Within this framework, business managers, business operating personnel, O-O project managers and developers can address the fundamental interactions between various elements of the development environment, thus assuring the creation of flexible, high-utility software components.

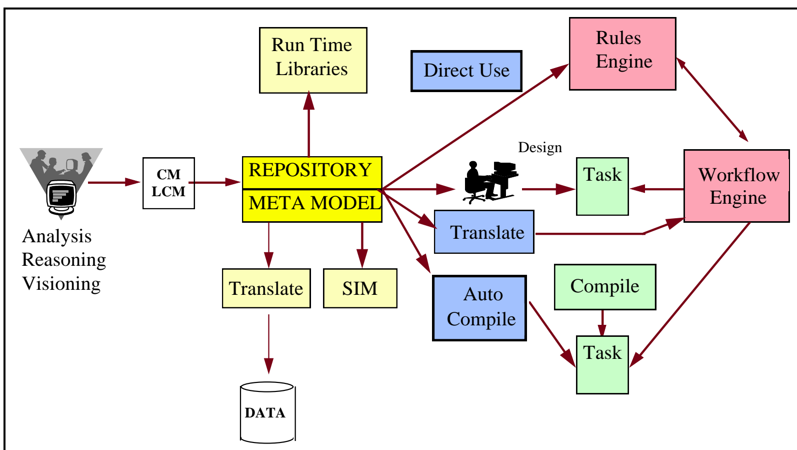


Figure 0-2

CM	Configuration Management	Translate - Design	Translate rules via manual design processes
LCM	Life Cycle Management	SIM	Simulation Interface Model
Run-time Libraries	Static and dynamic link libraries	Direct Use	Run-time binding of rules to application
Repository	Commercial off-the-shelf package	Auto-compile	Recompile all dependent applications
Meta-model	Specification for repository structure and content	Rules Engine	Special-purpose or Commercial off-the-shelf package
Translate -DBMS	Translate rules to DBMS components + SQL	Workflow Engine	Commercial off-the-shelf package

Business Person's View

Most of the rules that function in the business rules environment will be the kind of dynamic rule that changes when its underlying business policy and/or procedure changes. It is vitally important that the rules environment meet the requirements of the business as well as the technical requirements of rules management. It is fair to say that the business imposes two basic requirements on the rules environment.

First, the rules management component must have a good user interface. The interface must help the business manager and the rules manager to work together to solve problems. When the business manager proposes a rules change, both parties must be able to satisfy themselves that they have located all existing rules that might change as a result. Communication between these two managers must be unambiguous. The presentation must be non-technical. It is not reasonable to expect a business manager to learn special languages or graphic conventions in order to communicate business requirements.

Second, the rules management component must support the entire rules life cycle. The rules manager must be able to create and delete rules, generate source code from the rules created, compile the source code into object code, and adjust any run time parameters affected by the new rule. Both accuracy and the efficiency of the business rules system are highly dependent on having rules management software that a single rules manager can use through the rule's entire life cycle.

The Role of Rules in Business

When all is said and done, the rules in the business rules environment will be those that are most important to business process owners. The main reason to put these rules in a special environment is to give non-technical managers more control over the system behavior that they care about. Conversely, most rules governing purely technical aspects of an application system do not belong in this environment. Here are some of the main reasons for including rules in the rules environment:

Rules provide an index into the business application. It is all but impossible for a non-technical business manager to locate an instruction of interest in conventional application code. Of course, for poorly organized and documented code, this task is hard for programmers to do as well. Since rules have a consistent structure, it is possible to store their parameters in a database. It is then possible to query the database to locate the rule or rules in question.
Rules provide a consistent way to express operations. Even if the manager could find the code, it would still be hard to interpret. Rules reduce the syntax of any operation to a variant of the "WHEN...IF...THEN..." theme. That makes rules relatively easy to understand, even when the syntax is formal and technical.
Rules have English-like and graphic presentations. There is no compelling reason to express a rule in technical terms. Instead, it is best to frame rules in a language that is easy for managers to understand. We can show the workflow by using flow charts.

Rules "cut out the middleman." Maintenance of the rules base usually does not require a large staff. In most cases, the business manager need only work with one rules manager. The rules manager, in turn, can make most changes without assistance from other technical staff. There is less need for the old analyst/programmer/quality control division of labor and the loss of communications that goes with it.

Rules are easy to change. Because rules simplify the maintenance process, they shorten the response time for implementing changes. This allows management to change systems operations quickly to meet changing competitive needs.

Views of Rules

These business needs demand a carefully designed rules environment. From the viewpoint of the business manager and the rules manager, the interface to the rules environment should support four different views of every rule in the system.

The "business rambling." Business rules start out in life as what Barbara Van Halle and others sometimes call "business ramblings." These are usually the first cut at business rules gathered directly from business managers. They can be vague, incomplete, incoherent, contradictory, or redundant — sometimes all at once. However, it is important to preserve them. They are the basic documentation of any business-driven development project — the expression of business needs in the business manager's own words. Note that many formal and technical business rules may result from a single "business rambling".

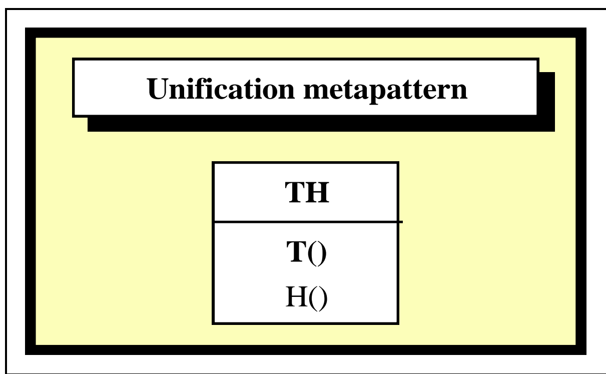
- The "English" view of rules.
  For most purposes, the best way to express a business rule is to use a formal, but English-like syntax; for example:
  ON COMPLETION OF Order's orderEntry ROUTINE,
  IF Order's TotalAmount is greater than $100,
  THEN EXECUTE Order's checkCredit ROUTINE.
- The rules browser.
  The repository contains both rules and the object model with which the rules react. The goal of this repository is to allow both business and technical managers to navigate the rules base. They must be able to find rules responding to selected events, triggering selected actions, or responding to selected state changes.
- The graphic view of workflow.
  Rules and processes define workflow. A trained programmer can visualize a workflow by looking at application code. A business manager without training in programming cannot. Business managers will therefore need a simple graphic presentation of workflow such as a flow chart.

Design Patterns of Rules-based Architectures

In this section we provide some substantive technical concepts for implementing rules in a flexible, durable way, since in our view this is often the most serious challenge to object-oriented application software development.

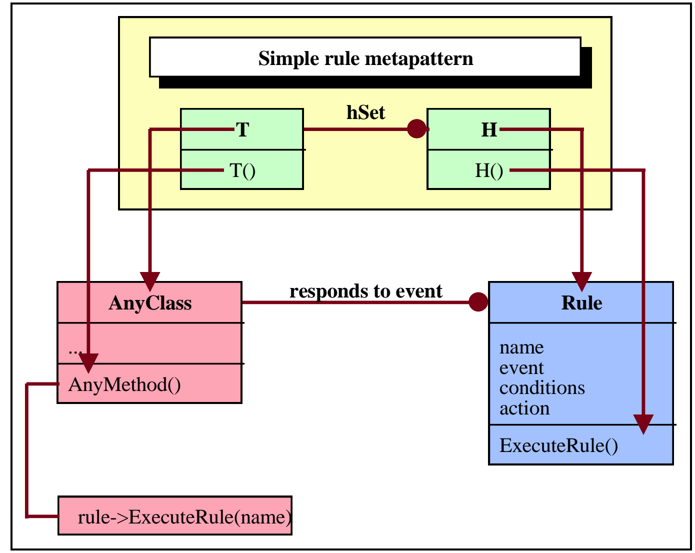
Though we subscribe heartily to the Design Patterns Catalogue approach of Gamma, et al., we find it helpful to match up business rule forms (taxonomies) to specifics of the design using Pree's template-hook exposition.

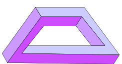
Pree calls the fixed part of an object group (or framework) a "template" and the variable part (corresponding to a hot spot) a "hook." Meta-patterns are simply different fundamental relationships among classes and methods. Rather than try to explain the technology, let us simply show a drawing, after Pree, of the simplest meta-pattern, the Unification meta-pattern:



In this meta-pattern, the metaclass **TH** has two methods. Method T() is the template method. Method T() never varies. Method H() is the hook method. It contains the hot spot code. Method H() is different for each class derived from metaclass **TH**. Since T() always calls H(), the developer can use the inheritance mechanism to create new methods, derived from **TH**, each with the identical T() method, but varying its behavior through different versions of H().

From this perspective, a rule is a kind of hook method. That is, a rule is a way of separating the variable portion of some behavior or process from the constant behavior. A Rule meta-pattern might look like this:





This is the simplest possible rule meta-pattern. The template class **T** can be any class at all. The hook class **H** is the rules class. The template method T() simply calls the hook method H() to execute a rule. In this simplest of all possible approaches, there is no need to create different versions of the rule class. Instead, you can simply treat events, conditions, and actions as parameters. The effect is to extract a tiny piece of logic from any class in the application where it is hard to find and work with, then convert it to data in the form of rule parameters where it is easier to find and work with.

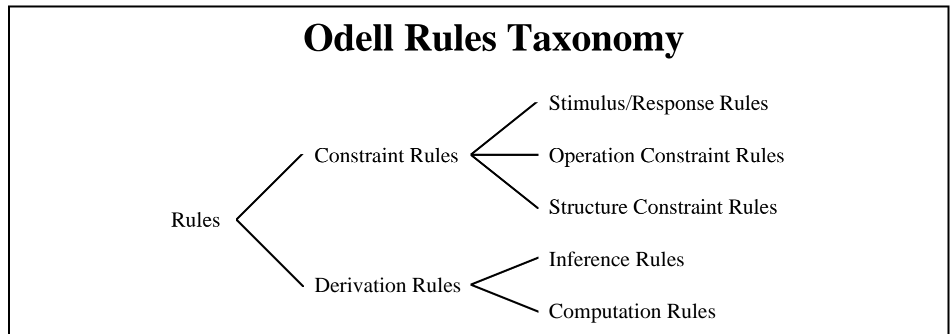
Rules Taxonomies

The rules management framework should be independent of rules design methodologies. Different rules managers have learned different approaches to the acquisition rules and the design of rules systems. The rules base should be flexible enough to catalogue rules by more than one rules taxonomy.

A rules taxonomy is a catalog of rules types. A rules type is a kind of design pattern for rules. Just as Gamma and Pree have developed catalogs of object design patterns and meta-patterns, rules experts have developed catalogs or taxonomies of rules types.

The choice of rules type depends on the designer's sense of what is constant and what varies. This is no different than the choice of design pattern. Different analysts will focus on different hot spots and come up with different patterns. Although we use the rules taxonomy of James Odell, any of the several O-O rules methodologies is sufficient to support the business rules management approach we describe here.

We will illustrate the point with the Odell taxonomy, which organizes rules into a tree with five basic types at its leaves:



Here in tabular form, are each of Odell's five basic rule types along with its description and a diagram of its fundamental design pattern, following the manner of Pree.

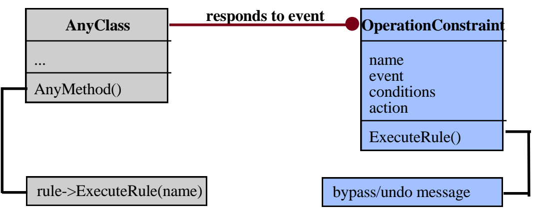
Stimulus/Response Rules

These are the archetypal ECA (event, condition, action rule). In all cases, an event triggers the rule, which evaluates a condition, then initiates an action. As defined by Odell[4], one specific event triggers these rules, which consider a set of conditions and then take one specific action.

AnyClass
...
AnyMethod()
responds to event
S/R Rule
name
event
conditions
action
ExecuteRule()
rule->ExecuteRule(name)

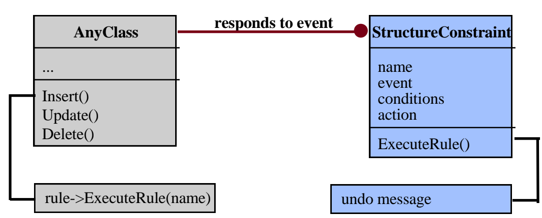
Operation Constraint Rules

These are similar to stimulus/response rules except for limitations on the scope of their action. There are two subclasses of operation constraint rules. *Operation precondition rules* specify conditions that must be true if an operation is to continue to completion. *Operation post condition rules* specify conditions that if not met cause the operation to abort and the object in question to return to its former state (for example a database rollback). Although there are a number of ways to implement this rule, we show it here returning a message to either bypass (precondition) or undo (post condition) the operation.



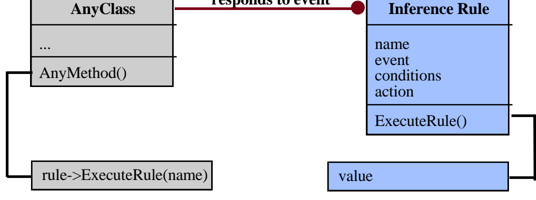
[4] James J. Odell, *Journal of Object-Oriented Programming*, May, 1993. This article applies to all reference in this table.

Structure Constraint Rules



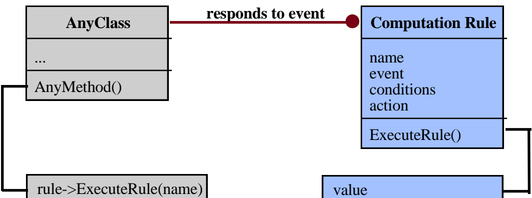
Odell states: "Structure constraint rules specify policies or conditions about object types and their associations that should not be violated." Database triggers are the archetype of these rules. Note that unless you have complete control over the process of modifying an object's state, you cannot guarantee the execution of a structure constraint rule. For example, the database guarantees that triggers fire in response to standard SQL events. In our illustration, we guarantee this control through the *Insert()*, *Update()*, and *Delete()* methods.

Inference Rules



Inference rules return conclusions if a given condition evaluates to true. In other words, inference rules return values. Here we illustrate an inference rule in the ECA form. Classical expert systems implement inference rules without any triggering event. Instead, a rules engine evaluates conditions and executes rules, as required, when the conditions are not true.

Computation Rules



These rules also return values. The difference is that they do not depend on any condition. In other words, the condition always evaluates to true.

Conclusion

In this chapter we have identified a major omission of most projects, and O-O projects in particular — that is, project managers and software developers often overlook the critical importance of application architecture in meeting the demands of the business users they serve.

We have offered a business rules approach to:

1) Improve communication with the business community,

2) Enable far more robust designs than currently available methodologies offer, and
3) Capitalize on the innovative design patterns developments that workers such as Gamma, et al, and Pree have spearheaded.

All of these elements are requirements for successful development in the O-O community. Without them we may achieve great excellence in applying the O-O paradigm and tool sets, yet still fall far short of the goals and objectives of our user community. With a business rules based Application Architecture, managers and developers can be confident they will be able to respond to the ever-changing needs of the business with effectiveness and efficiency.

management strategies

For thirty-three years, Management Strategies has worked with a number of Fortune 1000 companies in creating software solutions that save money, build profits, and streamline data processes.

Our extensive experience includes expertise in business process engineering, cost reduction efforts, information strategy planning, rules-based distributed architectures, and object oriented technologies.

369 Congress Street • Boston, MA 02210 • 617-482-3262 voice • www.ms-strategies.com