9

All the World's a Stage

Chapters 7 and 8 focused on individual objects and problem domains. The discussion focused on discovering, identifying, the objects that already exist in a domain and understanding what is expected of them in that domain. We focused on an entire domain – either an enterprise as a whole (XYZ Corporation) or an area of commerce (banking industry) as a whole – in order to avoid defining objects in terms of their idiosyncratic behavior in narrowly defined applications.

Our goal of object discovery was facilitated by telling specific stories about small groups of objects engaged in resolving specific problems or, collectively, providing a specific service. We did this to illustrate, to explore, and to discover the *intrinsic* capabilities and expectations of that object. No single story defined the nature of an object. The *set* of stories we use to discover and model objects were broad enough in scope to assure a proper balance of specificity and generality. Specificity - so that the objects could be built, and built simply. Generality – to

assure the same objects could be used in multiple contexts, without modifying their essential nature.

A number of concerns that typically arise in software development were not mentioned or were mentioned but set aside for later discussion. Deferred issues fall into four broad categories:

- Static relationships, associations, affiliations, compositions, and interaction channels that are assumed to be, but seldom are, permanent.
- Scripting, providing the cues to be used by a collection of objects while completing specific collective tasks.
- Constraints, keeping objects from doing things they are capable of, but which we wish to prevent in certain circumstances.
- Implementation, providing detailed information about the inside of our objects – algorithms to be used and detailed specification about formats and values of information.

The first three categories were deferred because they do not inform us about objects, only about contexts in which objects must operate. Implementation issues are deferred in accordance with the longstanding development rule – "figure out what must be done before you concern yourself with how it is done."

We are also being consistent with the Lego Block metaphor – separating the problem of *creating* blocks from the problem of building things *with* blocks. To reintroduce another metaphor – we separated our discussion of actors and their intrinsic talent from the discussions of casting, screenplays and direction. Our goal was to create actors (objects) with the versatility required to assume many different roles in a wide variety of genres and avoid creating objects

that were typecast – doomed to playing the same role, in the same screenplay, over and over again. (As an aside, typecasting works – i.e., provides a steady income - for some actors in Hollywood only because so many films are imitative rather than innovative. In software we want to be better than that.)

Generally, we do not try to solve all the problems of the world at one fell swoop. Instead we pick a particular problem in, or attempt to enhance a particular aspect of, a domain. Our focus might be as narrow as a single object with a specific behavior; or might encompass a single software program that coordinates a variety of behaviors, or might be several programs – what we typically call a system. Whatever the scope of our focus, we will use the term "application" or "application artifact" as an umbrella label. [To see the difference between application and application artifact, see sidebar – Systems and Artifacts.]

Systems and Artifacts

Structured analysis was, in its own time, a new paradigm for software development. As popularized by Larry Constantine, Ed Yourdon, and Meillor Page-Jones, an initial task was the creation of a model of the "existing system." Developers were instructed to create a model – data flow diagrams, entity relation diagrams, context diagrams, and even program structure charts – to document their understanding of how things were currently done. This effort was supposed to facilitate understanding and modeling of a new "system" to replace the existing one. In practice this step was seldom completed – its value was not understood and its cost was significant. Later editions of books on structured analysis tended to eliminate this step altogether.

Modeling the existing system has a parallel in object thinking – domain centric analysis. There is an implicit assertion in both: "you cannot begin to understand what must be until you understand what is." This assertion has two corollaries: one, "almost all of the objects you will ever need are already defined, and already have behavioral expectations associated with them, in the domain; and two, almost all the requirements new development arise from a misalignment of behaviors and objects. Misalignment results when the wrong object (or group of objects) is providing a particular service, a service is more appropriately provided by a silicon-based object simulation instead of a carbon-based biological object, or, occasionally, there is no existing object capable of providing the needed service.

Christopher Alexander (*Notes on the Synthesis of Form*) offers another expression of the need to understanding the problem and problem (domain) space, in detail, before proceeding with development. Design, according to Alexander, is the process of conforming a solution to a problem. The problem defines the needed solution and understanding the problem therefore "reveals" the required solution.

Object thinking and structured analysis share a common belief that the existing "system" must be understood before proceeding with development. They differ, however, in a very significant way. Structured thinkers – like software engineers – tend to think of the "system" in terms of a bounded artifact, a piece of software executing on hardware. Object thinkers view the "system" as *The System* – the complex whole of the domain and all its parts (objects) – explicitly including all the human objects – and all of the patterned interactions of those objects. It is this System that we seek to understand before attempting to intervene with any development project.

Understanding an entire System is, seemingly, a daunting task - one reason why most software engineering texts suggest a much narrower focus. Paradoxically, expanding our focus actually provides significant benefits and has the effect of simplifying our work. The dictum, "Everything is an Object," provides us with a single metaphor/model and a means of partitioning even the most complex domain into a relatively small number of objects with easily understood behaviors.

This perspective also has the effect of redefining what is meant by "artifact" and "application" as those terms are used in development projects. Artifacts are objects or methods – nothing more. Applications are simply scripted assemblies of objects; some objects being software simulations, others being physical entities, and still others being human roles.

A side effect of this perspective is a kind of "minimal intervention" principle. Recognizing that introducing any change into a complex system will have widespread, frequently unforeseen, and too often deleterious consequences, we seek to minimize those effects by making the smallest possible change. For example: introducing a single new object, adding a new behavior to an existing object, reassigning a behavior from one object to another, simulating as faithfully as possible an existing object with software, or assembling and scripting the least number of objects necessary to fulfill a single story – as XP defines user stories.

We do need to understand more about the application – the context in which our objects will perform and the precise tasks we expect our objects to perform – before proceeding with development. Specifically we need to determine what relationships, constraints, and interactions

will be required of our objects in each specific circumstance. (Although we might discover new things about our objects while engaged in our study of the application and might be required to modify the intrinsic nature our objects as a result; it is a major mistake, from an object thinking perspective, to define objects in terms of an application instead of a domain.)

Static relationships

There are an open-ended number of possible relationships among objects. Some are so common as to have special notation invented for them in modeling tools (e.g. UML): 'using,' is-a-part-of,' 'is-a-kind-of,' and 'contains' are examples of common relationships.

From this large set of possible relationships, only two tell us anything about the intrinsic nature of our objects. All of the others provide information about the context – the application - in which our objects operate, they are "situational relationships." The two relationships that describe intrinsic aspects of objects are; "is-a-kind-of" and "collaborates-with."

Is-a-kind-of relationship

Classes are not essential to object thinking, merely a convenience. From a cognitive standpoint, the use of classes allows us to think about an entire group instead of all the members of that group – something we have been doing throughout this book.

Combining the idea of classes with the "is-a-kind-of" relationship yields a taxonomy, a particular kind of structural organization, that can then be used as a kind of "index" to help in

finding an object with a particular set of behaviors. Figure 9.1 is a class hierarchy diagram – specifically, a taxonomy of the classes in the mortgage trust example.

F09xx01

Figure 9.1 – Class Hierarchy Diagram
Shows the is-a-kind-of relationships among the objects (via their classes) in the Mortgage Trust example.

Using a taxonomy requires only a general notion of what behavior(s) you need in an object as a starting point. You can find an object with that (or some of those) behavior(s) and then move up or down the taxonomy to find an object that matches your needs in a more precise manner. If not existing class of objects meets your needs you can still use the taxonomy to simplify your creative work by finding the object that best matches your needs and saying your new object is-a-kind-of the one you found. You then create a subclass and give it the precise behavior you want and use – via inheritance – behaviors that already exist in the super class.

From a cognitive point of view (we will discuss an alternative point of view in just a minute), your taxonomy *must* follow two rules:

1. Exhibit a single line of descent (single inheritance) based on nothing but the behaviors of the objects. Ideally, you should create your taxonomy (class hierarchy) as soon as you have completed side one of the object cube. That way you are not tempted to create a hierarchy based on object internals – i.e. methods or instance variables. Single inheritance is important to avoid "abominations" that break the taxonomy – like the platypus breaks the Linnean taxonomy of fauna.

2. Subclass only by extension – by adding behaviors. A subclass should be able to everything its super class(es) can do, plus at least one more thing. This means that you can substitute an object from a subclass in any situation calling for an object from its super class and still get the behavior you expect. If you define a subclass by subtraction (taking away a behavior) or by redefinition (overriding a behavior) you create cognitive dissonance. Users of your taxonomy can never be entirely sure of the behavior they will get from classes without looking inside of those classes to see what they really do. This is, of course, and egregious violation of encapsulation.

Class hierarchies can be viewed from an implementation perspective as well as a cognitive perspective. Unfortunately, these two perspectives are not always in concert. When they disagree it is almost always because the hierarchy is perverted to conform to constraints imposed by a non-object machine or virtual machine (programming language).

In Chapter Five – Vocabulary, Words to Think With, it was noted that one aspect of a class was to act as a repository for code and information shared by all instances (objects) of that class. In terms of efficiency (storage space is minimized and changes can be made in a single place) this is a good idea. It becomes very tempting, however, to use this fact to justify creating your class hierarchy based on shared code instead of shared behaviors. A similar error arises from treating object variables (instance variables) as if they were data attributes and then creating your hierarchy based on shared attributes.

A class hierarchy creates coupling among objects in each line of descent within that hierarchy. If you base the hierarchy based on implementation (shared code or shared "attributes") the hierarchy will be unnecessarily brittle – because implementation decisions change with much greater frequency than the real world. New technology equals new implementation decisions; new environment or implementation language, same thing. Technology changes on a roughly six-month time scale. The real world changes on an evolutionary timescale. If your objects truly reflect the natural world, they too should change on an evolutionary timescale. (Evolutionary in this case means something between biological evolution (eons) and socio-cultural evolution (generations), or, worst case, the pace of change of a business domain.)

Collaborates-with relationship

Collaborations arise if, and only if, object-A must use a service from object-B during the interval defined by object-A receiving a message (request for service) and object-A returning the appropriate result object to the sender of the message to object-A. All collaborations are synchronous in the sense that Object-A must invoke and wait for results from Object-B before it can complete whatever work is required in response to the message it itself received.

Collaborations are almost always "hard-coded" – they require manifest references in the algorithm of the method where the collaboration occurs. Because of the hard-coded nature of collaborations we assert they define part of the intrinsic nature of an object. In every circumstance, in every context, the collaboration will occur if object-A receives the indicated message.

Collaborating objects are very tightly coupled. For this reason, collaborations should occur inside of the encapsulation barrier – with objects occupying instance variables, objects received along with messages, and objects occupying temporary variables. Collaborations with objects occupying global variables are good or bad depending on the justification of need for a global variable. Collaborations outside the encapsulation barrier are sometimes necessary but should be avoided (by refactoring the distribution of responsibilities among objects) whenever possible.

Wirfs-Brock introduced a dedicated model – a collaboration graph – that was seldom adopted by actual developers. Figure 9.2 shows an example of such a graph. The graph depended on the use of contracts (side three of the object cube) to define potential connection points for potential collaborations. Deciphering the graph was difficult because of line tracing from client to server. Collaborations graphs (probably unintentionally) provide a visual metaphor consistent with the object = software IC introduced by Brad Cox. Interesting collaboration graphs look like circuit boards when viewed as a gestalt.

F09xx02

Figure 9.2 – Collaboration Graph
A visual model of collaborations – based on graph appearing in Wirfs-Brock 1991.

Situational relationship – Static Relationship Model

Collaborations are not the only example of "uses" relationships among objects. It can be said that object-A uses object-B if object-B had asynchronously provided information or services to object-A that object-A finds useful for its own work. For example; an airplane has a responsibility to identify itself, which requires an id object, which leads to the assumption that at some point in the past the airplane asked some national naming authority for an id object, and that request for an id object is an example of a uses relationship between airplane and namingAuthority. (It is not a collaboration because the airplane does not interact with namingAuthority each time it receives the id self request.)

Which objects use the services of others – outside of collaborations – reflect the demands of the application (the situation) and not intrinsic needs of the objects themselves. The same thing can be said of which objects are aggregated into components – the aggregation is a reflection of the demands of a particular circumstance, nothing more. This is not to say that it is unimportant to understand (and probably model) situational relationships – but it is important to recognize, and keep in mind, that you are not modeling the intrinsic nature of any objects.

What then do you need to understand about the application? What models might be helpful as a guide to thinking about your application?

- What objects are participating in the work of the application?
- What objects have a need to communicate with others and what is the general nature of that communication?
- The "data" – really the collective memory of objects involved in the application – any relationships among data items

· An overview, or gestalt, view of the application and its participants.

A static relationship model (UML equivalent is a class relation diagram or an object relation diagram) can provide all of this information. Static relationship models (SRM) are models of applications and not domains. They are called static because the information portrayed on this type of model is supposedly invariant. It must constantly be remembered that invariant applies to the time and space boundary defined by each individual application – the relationships may or may not exist in any other application. This is a major point of difference with data-driven approaches to modeling objects.

A data-driven approach assumes that objects are equivalent to data entities. Data entities *are* expected to be defined at the level of the domain because it is assumed that they are the basis for creating a database which will be common to all applications. This is one more example of perpetuating the traditional view of software as "data structures plus algorithms." Data-driven approaches to objects posits a singular database containing all the data structures and allows multiple applications –application = algorithms only – to share a common data structure.

An SRM, from an object thinking point of view, is a gestalt view of the objects participating in an application (cast of characters) and relationships that exist among those objects when in that application. Relationships most frequently denote *potential* interactions among objects just as a roadmap shows *potential* travel[1] among cities connected by those roads.

Relationships might also denote constraints on object behaviors or associations while participating in a given application.

Most application SRM diagrams are little more than semantic nets with more precision, more detail, and with an aura of specification instead of discovery. A semantic net also identified the cast of characters – objects - in a domain and revealed relationships among those objects. Semantic nets were used as a heuristic starting point for object discovery. SRMs are intended as a kind of specification – information to guide the developer in the creation of the application.

Most of the information depicted in a semantic net will be used to define and create objects. Some of that information will be used to define applications, especially constraint or rule based information on the semantic net. Consider the mortgage trust example discussed in previous chapters. Figure 9.3 shows a fragment of the semantic net dealing with relationships between the price of a home and the median price of homes in a neighborhood. The semantic net fragment shows a specific business rule – the home price must be less than or equal to the median price of the neighborhood. This rule is not a domain rule – it is an application rule. It should not become a basis for defining the home, neighborhood, or price (actually money) objects because it does not reveal the intrinsic nature of any object. Instead it shows a relationship that exists only in one or more applications (contexts), perhaps the "collect and validate loan application data" application.

F09xx03

Figure 9.3 – Semantic Net Fragment
Shows a constraining relationship among objects in a domain.

Figure 9.4 shows a SRM for the mortgage trust example. Note that it is very similar to the semantic net – most of the same players are still present and most of the links among them are still evident as well. It differs from the semantic net in the use of implementation names for many of the objects (classes), it shows some of the interesting details of object construction (instance variables and method names), and it uses terminology more suitable for implementation (technical rather than domain jargon is allowed on an SRM but not a semantic net) than discovery.

F09xx04

Figure 9.4 – Static Relationship Model
SRM depiction of the classes and relationship in the mortgage trust example.

Situational Relationship – Collective Memory Map

One of the important responsibilities that an object might assume is to portray and maintain some specific bit of information (data). Examples include numbers, integers, floating point numbers, dates, times, characters, strings, and money. None of these objects limit themselves strictly to the portrayal and maintenance of information, they also have other interesting behaviors appropriate for the kind of information they maintain. Dates, for example, can perform calendar related calculations and comparisons. Numbers can do arithmetic. Strings have many behaviors similar to collections – iteration, insertion, deletion, and arrangement, for example.

Given their specialized nature, "data depiction objects," frequently are found occupying instance variables of other objects. An airplane object, for example, has an instance variable called 'id' which is occupied by a string object.

Given the information centric nature of the contemporary world, numerous constraints and relationships exist between and among data depiction objects. Although it is possible to capture this type of relationship in a standard SRM, doing so obscures, somewhat, the real nature of the relationship. Suppose, for example:

- An airplane object has instance variables for: **id**, containing a serial number like N543UE; **transponder code**, a four digit integer; and **status** containing a character whose value indicates that airplane is in normal operation, hijacked, declared emergency, or in controlled airspace.
- An airspace object has an instance variable, **assignedAircraft**, that contains a collection of "blips."
- A blip object has instance variables for **id**, **transponderCode**, **altitude**, **latitude**, **longitude**, **vector**, **priority**, and **status**.

F09xx05

Figure 9.5 – SRM Fragment
Relationships among airplane, blip, and airspace.

Figure 9.5 shows a fragment of an SRM that relates the three objects under discussion. This fragment obscures a great deal of detail about the actual relationships involved. For example, the following rules:

- Transponder code value must equal 1200 for VFR flight.

- Transponder code value must equal 7500 if plane has been hijacked.
- Transponder code value must equal 7700 if plane is in a state of emergency.
- Transponder code value must equal 7600 if plane has lost communications.
- Transponder code must equal value specified by ATC for its blip representation.
- Transponder code value must be consistent with airplanes status value.
- Blip id and airplane id must be consistent.
- Blip priority must be '1A' if transponder code is 7500, 7600, or 7700.

It would be possible to model all of these constraints simply by adding relationship symbols between blip and airplane on the SRM – if you are willing to sacrifice a great deal of readability. Alternatively, since all of these relationships are among objects occupying instance variables of other objects there is value in modeling them directly in a separate model, a collective memory map like that depicted in Figure 9.6.

F09xx06

Figure 9.6 – Collective Memory Map Fragment

Depicts relationships among objects occupying instance variables of other objects in the airplane, blip, and airspace example.

A collective memory map (CMM) is another type of static diagram constructed according to the following rules:

- Objects appearing on a CMM are occupants of instance variables in other objects.
- Objects appearing on a CMM must have a primary role of data depiction and maintenance – they can be considered "primitives" in the same sense that a database defines certain objects

as primitives. Most often these include numbers, characters, strings, dates, money, time, and similar objects.

- A kind of collection that is restricted to containing "primitive" objects, called a ValueHolder, can also be depicted in a CMM.
- Each object depicted is named using dot notation that identifies the container object name and instance variable name, e.g. *customer.lastName*.
- In those cases where the instance variable contains a ValueHolder instead of a primitive, the dot notation is extended to include the label (key) associated with the actual primitive, e.g. *customer.description.gender*.
- Relationships among objects on the CMM are drawn as connecting lines with appropriate labels.

Figure 9.7 depicts a collective memory map for the mortgage trust example.

F09x007

Figure 9.7 – Collective Memory Map
Mortgage trust example.

Collective memory maps are situational – they reflect constraints that exist among "data" objects in the context of a particular application. In some cases it might be useful to create a domain level version of a collective memory map. A domain level collective memory map would have some things in common with an enterprise memory model – i.e. a global depiction of all the "data" in the domain and the objects in which it might be found and any relationships among those objects that were truly invariant across the domain. If your enterprise is interested in constructing such a global model you need to remember some important caveats:

- Object thinking presumes that "data" objects represent information or knowledge that objects need to perform their assigned tasks while enterprise data models presume to depict all the data that the "system" must remember about objects. This is a critical distinction.
- The global model should contain only relationships and constraints that are invariant across the domain.
- A CMM for an application may add, delete, and modify relationships to reflect the situational context. If there is a conflict the global map the global must be modified.

Situational Relationship – Architecture

Almost all applications with have an overarching structure or means of organization – that might be called "architecture." Architecture affects the way you think about implementation, especially when the architectural pattern is so common as to be assumed. For example, traditional structured development presumed a hierarchical command and control architecture as reflected in a program structure chart, Figure 9.8, This visual pattern is seen in the mainline-plus-subroutines source code organization scheme, Figure 9.9. The hierarchical-control architecture is a codification and expression of a lot of ideas about "good" program design, ideas that support most of the rest of structured development approaches.

F09xx8

Figure 9.8 – Program Structure Chart
Depicts the proper organization (architecture) of a program consistent with structured development ideas.

F09xx9

Figure 9.9 – Source Code structures
Shows how actual source code might reflect the hierarchical control architecture

A prototypical architecture that reflects object thinking ideals and principles also exists – Model-View-Controller (MVC). MVC is not a mandated architecture – you can use almost any of the commonly encountered architectural patterns (see sidebar, Architectural Patterns and Objects) but an investigation of MVC is instructive and important for the way it illustrates and reinforces the other aspects of object thinking that have been discussed in previous chapters.

Architectural Patterns and Objects

MVC is but one of a number of architectural patterns or prototypical ways of organizing a group of objects engaged in a collective task. Commonly encountered alternative architectural patterns include: hierarchical control, pipes and filters, client-server, and several variations of the blackboard architecture. With the exception of hierarchical control (the antithesis of object ideals), all of these alternative architectures can be consistent with the principles of object thinking.

Pipes and filters – readers familiar with Unix/Linux will immediately recognize this pattern. There are two kinds of objects involved – pipes which are connectors and, possibly, temporary storage locations; and filters which are places where very specialized services (transformations) are performed. Figure 9s.1 illustrates a small pipes and filters architecture.

9s09xx1

Figure 9s.1 – Pipes and Filters
Objects move along the route provided by the pipes and obtain services from the objects embodying the filters.

Traditional thought about pipes and filters architecture reflects traditional thinking about the separation of data and process. From that non-object point of view, data passes along the pipes (pushed or pulled by the connected filters) and is manipulated by processes at each filter station.

Simply reversing the idea about who is in charge and assuming everything is an object, makes this a perfectly acceptable architecture for use by object thinkers. The architecture merely provides transport objects (pipes) and service providing objects (filters). A large pool of objects requiring a similar set of services can ask the pipes for transport to the filters whose services they need, take advantage of both and effect self-processing in a very efficient manner.

Client-server architectures are the mainstay of the networked business world. Two tier client-server systems are almost identical with MVC with the model (usually a database) physically located on a server and most of the view and control aspects on clients. Conversely, MVC can be seen as simply an n-tier (or peer to peer) client server architecture. Object thinking will frequently, and naturally, lead to the construction of a client-server architecture – the only difference being the overall distribution of responsibilities and the absence of overt controller, manager, and scheduler type objects.

Blackboard architectures actually come in at least three variations, which we will label: bulletin boards, blackboards, and whiteboards. All variants have common elements as illustrated in Figure 9s.2, i.e.: service providers, requestors, a common communication space, requests for service, and results. Each of these elements is, of course, an object.

9s09xx2

Figure 9s.2 – Blackboard
Common elements to all blackboard type systems.

The simplest form – a *bulletin board* – requestors post requests in the common space, service providers constantly poll that space looking for requests they might service. Upon finding an appropriate request the service provider retrieves it and provides appropriate satisfaction, a result, which is then returned to the common space where the original requestor finds it (constantly polling the space to see if it, "is there yet?") and retrieves it.

Request and result objects are relatively straightforward containers – of the information necessary to specify a result and the result which can be queried by the requestor upon its retrieval. A ride board, like that found in most university student unions, is a prototypical example of a bulletin board type system.

Give the common space some interesting responsibilities and you have a *blackboard* variant. The common space maintains a list of service providers to which it can match a posted request. When a match is found the request is forwarded appropriately. The common space also retains a list of requestors and their requests so that, when a result is posted it can forward the result to the appropriate requestor. Care must be taken not to turn these responsibilities into a command and control structure.

Whiteboards are similar to the "tuple spaces" proposed by David Gelernter (noted MIT computer scientist) and implemented in his Linda programming language. In this variant the request and result objects are given enhanced responsibilities and the common space becomes a work space (where requests and results can interact with each other) as well as a communication space.

All of these architectures are perfectly consistent with the ideals of object thinking – as long as the assumptions about control, about distribution of responsibility (factoring) and object autonomy illustrated in MVC are respected.

Figure 9.10 depicts the model-view-controller architecture. The figure is actually a slight variant of the architecture proposed as the Smalltalk language and development environment was being created at Xerox PARC. MVC is almost always discussed in the context of a graphical user interface environment – which being invented simultaneously with Smalltalk – but the important principles behind the architecture are not limited to such an environment.

F09x10

Figure 9.10 – Model-View-Controller Architecture
Generalized model of the MVC concepts.

MVC is grounded in the idea of specialization and distribution of responsibilities; specifically, the identification of three main categories of responsibility (presentation or visualization, computational work, and coordinating communications) and assigning those to different sets of objects.

The Model is comprised of those objects actually engaged accomplishing the work objectives of the application. Model objects have a need to communicate with each other by sending messages and by notifying other objects of internal state changes. In Figure 9.10, model objects are depicted as round circles.

One of the model objects assumes the role of the Application Object. The application object assumes responsibility for startup and shutdown activities – in collaboration with the other model objects – and acts as a kind of global (within the application) repository of objects and information that need to be visible to other model objects while the application is running.

The View consists of a hierarchically organized collection of objects – e.g. widgets, windows, icons – whose primary task is creating visualizations of objects so as to be comprehensible to human beings. This definition presumes a GUI environment and reflects the history that shaped the original MVC architecture. A view object does not have to be a GUI element. If an object creates a representation of itself other than its "native" implementation, that too would be a view object. Examples of non-sensory views would include bit stream serializations that allow objects to exist in a relational database or an XML statement that allows the object to be shared across applications and implementation languages.

In Figure 9.10, view objects are depicted as rectangles. The hierarchical organization of views reflects that fact that very simple objects have very simple views and more complicated objects have views that are collections of the views of the simpler objects of which they are comprised. It is also possible for any object to possess multiple views of itself. An integer, for example, might have a simple bitmap view that depicts the value of the integer and it might have an "update view" consisting of a widget depiction (rectangular entry field area) plus the bitmap of its value. Figure 9.11 shows one example of how hierarchical composition of views yields a complex view.

F09x011

Figure 9.11 – Composite View

The application completion screen in the mortgage trust example is a hierarchically organized collection of object views.

The Controller is badly misnamed. The "control" word is anathema to object thinkers and right thinking autonomous objects everywhere. We will substitute the word "Coordinator" for the rest of this discussion. Coordination refers to the tasks involved that allow one object to send a message to another and for an object to notify any interested object of a state change in itself.

Assuring that it is possible for two objects to exchange messages is a matter of visibility – the objects have to be able to see each other. An example of coordinating visibility is for the application object to have a variable that contains a collection of all the objects participating in the application. Should one object need to talk to another it can ask the collection for an object by name or by criteria. Similarly, the application object might have a variable which will contain the "currently active" or "currently selected" object to which other objects may need to send messages. Because the application object is global within the application, all other objects can send messages to it – or to the objects occupying its instance variables.

Event (state change) notification was the main responsibility of historic controller objects – another manifestation of the GUI-centric history of MVC. We will use a special purpose object, an eventDispatcher, to provide all the necessary coordination associated with event notification. Dispatchers were introduced earlier, but as a reminder:

- An eventDispatcher is a simple two dimensional table, the first column of which contains a list of events that can be dispatched and the second column of which contains a collection (possibly an ordered collection) of notificationRequest objects.
- A notification request is a dyad consisting of the name/id of the object requesting notification of an event's occurrence and a message to be sent to that object as means of notification. If the notification collection is ordered – e.g. for priority – the notification request may be a triad of name, message, and priority number.
- An eventDispatcher may add or delete events, must accept and delete registrations on request, and when an event is detected must tell each registration in the associated queue to execute itself (send the contained message to the contained object reference).
- If the registrationQueue is an ordered collection, it must re-order itself each time a registration is added or deleted.

EventDispatchers are depicted in figure 9.10 as triangles. Each object has a small solid triangle inside of itself, indicating that every object has the potential capability of accepting registrations for changes (as indicated on side six of the object cube) within itself.

Triangles appearing between an object (circle) and a view (rectangle) coordinate and assure consistency between the object and its view. Just as views are composites, so too are these event dispatchers. A composite object (Customer for example) might have a composite view consisting of the views of each of the objects occupying an instance variable of customer (the strings in fName, lName, and MI for example). Although it is possible for each object participating in this composition to use its own individual dispatcher to coordinate changes, it is also possible to create a single dispatcher and have each object add an event line to that dispatcher as it adds its view to the composite. Figure 9.12 illustrates this composition.

F09x12

Figure 9.12 – composite view and dispatcher

The composite object, Customer, might use a composite dispatcher instead of each object using its own dispatcher.

In addition to the explicit elements of MVC we also assume an 'Outside World' consisting of all the objects outside of the application object. This is the realm of human being objects, legacy system objects, and operational platform objects like hardware, operating systems, mouse managers, and keyboard managers. (Of course, any object whose name includes the term 'manager,' as part of its name was not designed by object thinkers.) Although it is true that everything is an object it is equally true that many of the things in this realm operate in a very non-object fashion.

From the perspective of the application object and the objects comprising its cast of characters, the outside world is a source of information and events. It is the presumed destination of results but whether or not someone actually uses the results an object was asked to provide is not of much concern. Objects live to serve, someone on the outside asked for a service, and it was provided. The object assumes that the client knew what it was asking for and had a use for the result. This is an anthropomorphic way of saying that objects do no and should not be aware of their clients, even when those clients are not other software objects.

The application object, or one of its cast members, may require services from objects in the outside world. In that case they may need to display themselves using an update view – an implicit request to an outside world object to provide information by altering the update view. In

other cases they may need to find and send a message to an object in the outside world – usually by sending a message to the operating system object via the application object.

A composite eventDispatcher associated with the application object also handles events generated in the outside world of which objects inside the application (or the application object itself) may need notification. In Figure 9.10 the application dispatcher has three parts (view-object coordination events, outside world events, and global notification events) illustrating its composite nature.

The MVC architecture may be flawed, as an architecture, but it is an excellent example of how to organize and coordinate a group of objects without imposing any kind of centralized control. The distribution of responsibilities and the creation of special purpose objects – like the dispatchers – allows complete flexibility, even at run time. Dispatcher events, for example, can be added and deleted by sending a run-time message as can registrations for those events. It is even possible to dynamically configure the event registration dyad at run-time. MVC provides an "existence proof" of how the ideals of object thinking can be realized.

Dynamic Relationships

All of the static relationships (except class hierarchy and collaboration) can be seen as "setting the stage" information. They determine the scenery and props, the 'marks' that actors use to navigate the stage, the cast of characters, and an outline of the plot. A script actually determines what happens on the stage during the performance of the play. A script is simply a collection of messages sent from actor to actor, interpreted by receiving actors, and resulting in

subsequent messaging. On occasion, actors respond to cues other than messages, signals, like the ringing of a phone, which is the phone's way of notifying other objects of a change in its state.

Capturing and modeling the dynamics of an application (following the stage metaphor) consists of detailed modeling of scripts and event notification – the two sources of "cues" for our participating objects.

Scripts

During discovery we told stories about object interactions. Some of those stories we wrote down on cards, others we modeled with interaction diagrams. Most of those stories exhibited a nested structure – analogous to the way a play is broken up into acts and acts into scenes. Design and implementation activities require that we turn those stories into specifications and/or program code.

The same tool, an interaction diagram, is used to discover conversational requirements and specify a script. The only difference will be in the labels attached to objects, messages, events, and returned objects. During discovery we want to use, exclusively, domain language in describing our stories. For design purposes, our diagrams should have labels that correspond to actual classes and actual messages as they appear in message protocols recorded on the object cubes. (In the Behavior! tool mentioned earlier, interaction diagrams have color coded labels, red meaning the label has not been mapped to an implementation term and reflects user vocabulary, black user vocabulary that has been mapped to an implementation term, and blue for

the implementation term. The diagram itself does not change – only the labels, assuring consistency between our domain and the simulation of that domain we are creating in software.) Figure 9.13 depicts the "discovery" and "script" versions of a conversation from the mortgage trust example.

F09x13

Figure 9.13 – Two versions of an interaction diagram

The diagram to the left has labels reflecting domain vocabulary. The one to the right is the exact same conversation with labels reflecting implementation vocabulary.

Extreme programmers spend time factoring their stories into discrete cards with the goal of creating a story card properly scoped for direct implementation. No intermediary design is required. An XP user story maps to a discrete bit of functionality of a larger program or system and should probably involve fewer than six objects exchanging ten or so messages. Purely as a thought organizing aid, even XP developers might want to take the time to sketch a rough interaction diagram as a means of factoring stories and as a prelude to coding.

For example, consider the story card, "application form is completed and validated," a story that almost certainly needs to be factored. Figure 9.14 a-c shows rough sketches of interaction diagrams that reveal nested stories and an outline of how the story can be completed.

F09x13a

Figure 9.13a – Interaction Diagram Sketch

The original story that, whenever it looks like a lot is going to happen, is factored with a placeholder for a sub-story inserted.

F09x13b

Figure 9.13b – Interaction Diagram Sketch
Sub-story, "entry field obtains and validates contents."

F09x13c

Figure 9.13c – Interaction Diagram Sketch
Sub-story, "application form validates self"

Note that the stories factored out of the original also yield sub-stories – this time mostly dealing with error handling.

Object developers wanting to design and model large scale conversations with multiple nested sub-parts (complete Use Cases ala UML) will use the same interaction diagrams with the difference of using a formal tool instead of a pencil sketch.

A final note on scripts: it is quite possible, and sometimes desirable, to implement scripts as first class objects, as an ordered collection of messages. When told to execute, the script sends the messages it contains, in order; accumulates the results; and, keeps track of its own status in terms of its successful execution. This approach adds "discipline" to a conversation, assuring that it occurs as it was supposed to, without adding "control." The script object encapsulates a conversational fragment and it is the script object itself that succeeds or fails in its assigned task. As a first class object, a script can be modified by adding and deleting message objects without the need to recompile the application containing those script objects.

Event Dispatching

We have two sources of information about object state changes as a source of cues for our scripts: side six of the object cube and interaction diagrams that actually place state change

detection and subsequent messaging into the context of a story. The object cube tells us of what state changes an object deems of potential interest to others and the interaction diagrams note occasions when state changes affect the flow of a conversation among objects.

The interaction diagrams also confirm an important aspect of object thinking – the only object directly aware of a state change is the object in which the change occurs. The double headed arrow used in an interaction diagram to denote a state change signal must always be directed to the object in which that state change occurs – never to other objects in the diagram. Figure 9.14 illustrates this important caveat.

F09x14

Figure 9.14 – Event notification in an Interaction diagram

State change notification is always a "message" to oneself – never to other objects. A state change can trigger multiple messages to other objects, reflecting the contents of the event dispatcher, as shown in the left hand figure.

All event notification is performed via an event dispatcher located in each object. Event dispatchers must be "populated" in order for an application to execute, i.e., each must contain a list of events and must have registrations[2] for those events consistent with the needs of the application as expressed stories (interaction diagrams). Because most objects will have little, if any, interesting state behavior, populating an event dispatcher will be relatively straightforward.

However simple an object's state behavior may be, it would be nice to have some corroboration of the information on side six of the object cube and in our stories. Taking the time to sketch a state model can provide this verification. State diagrams are also useful for

modeling (revealing) constraints on object behavior (see following section, Constraints) and for modeling event-driven (stimulus-response) scripts. A state chart, as devised by David Harel, has become the modeling tool of choice in the object community (albeit, most methods use a subset of the full Harel notation).

State Modeling

Figure 9.15 shows a state diagram for a mouse and part of a diagram for a "mouseManager" object. The diagram reveals that a mouse really consists of two concurrently operating objects, a trackball (movement detector) and a button. Both objects have very simple state but the diagram confirms what events need to be listed on the object cube for each object – 'click' and 'broken' for the button, 'newXYZ' and 'broken' for the trackball.

F09x15

Figure 9.15 – Harel State Chart
Chart depicts a mouse and it s composite parts plus a fragment of a mouse manager.

Inclusion of the mouseManager diagram fragment illustrates how object thinking affects distribution of work. Instead of making a complicated mouse capable of generating events for all possible state change combinations, we keep the mouse simple and have other objects assume responsibility for transforming simple events into composite events – in this case the generation of a 'double-click' event.

F09x16

Figure 9.16 – Harel State Chart
Partial charts for application and entry field, from the mortgage trust example.

The partial state chart shown in Figure 9.16 shows partial state charts for a mortgage application and an entryField. Assuming that the entry field in question is one of those contained on the form, the diagram reveals event registration information that needs to be accounted for and a constraint on the behavior of the application itself:

- Any change in the contents of an entryField object forces the containing application into the un-validated state. This means that the entryField object cube must contain a 'changed' event and that the application must register for notification of that event with all of its contained entryFields.
- The application cannot be funded if it is in the 'un-validated' state, reflected in the fact that the funded trigger only operates while the application is in the validated state.

Unless the developer is attempting to model a complex event-driven composite object – a graphical user interface perhaps – most state modeling is used only for the purpose of helping us think about possible constraints on an object's behavior, to confirm we have listed all necessary events on each object's cube, or to confirm/reveal the need for one object to register for events (state changes) with other objects.

Constraints

We have a stage, actors, and a script (application context, objects, and a script object plus populated event dispatchers) and the play (application) is ready to open. Except … sometimes our actors have to obey rules or are otherwise constrained in their actions. We need to accommodate this need as well.

Objects, like people, may have behaviors that should not be exercised in certain circumstances. All constraints are local and arbitrary (e.g. the rules of culture that allow one behavior in one place but not another) and, because of this, the modeling and the implementation of constraints are not intrinsic to the definition of objects, only to the context (application) in which those objects perform at any given time.

Classical thinking about object development made this difficult, primarily because rules and constraints were primarily seen, and modeled, in terms of static relationships between or among objects. Because these relationships were, supposedly, static they tended to be "hard coded" in object designs and implementations.

With a few exceptions, object thinking suggests that constraints and rules should be thought of as objects in and of themselves. We have talked in this vein throughout the earlier chapters and will make the idea explicit in the next section, Self-Evaluating Rules.

Constraints on objects can be direct, inhibiting a behavior (actually allow or prohibit the execution of a method) or setting limits on the values of objects contained in instance variables (the knowledge objects work with when doing their jobs). Constraints can be indirect, reflecting relationships among objects instead of object internals.

Examples of direct constraints include:

- Inability to respond to a message because the object is already engaged in performing work in response to a previous message.

- Limiting the set of objects to which an object will respond when sent a particular message. For example; the airplane would allow no one except a 'manufacturer' object to send the message that sets its id (serial number).
- Restricting the values that may be assumed by an object – telling an integer that it may only assume values between 10 and 100 would be an example.

An indirect constraint is illustrated with the following example: a marriage object must associate two, and only two, person objects. (Of course this is a local or cultural rule, not a universal, and so does not reveal the intrinsic nature of a marriage object.)

Constraints are revealed at different times in the development process. Stories, told to identify objects, are the most common source. Static models (discussed previously) reveal constraints that might have been assumed in the stories without explicit mention. Thinking about implementation reveals many others, especially constraints on values that objects may assume and state-based constraints.

Implementing constraints must be done with program code. That code is generally located in on of three different places: in the affected method, in a manager/controller object, or encapsulated in a rule object. The preferred option is using a rule object. Hard coding makes objects brittle and mandates undue maintenance if changes in constraints occur over time. Manager / controller objects violate object thinking principles (and from a purely pragmatic point of view, result in unnecessary complexity). Creating a class of objects capable of realizing any kind of rule – production rule, business rule, formula, etc. – is the object thinking solution to the constraint problem.

Self Evaluating Rules[3]

> *A Dependent is eligible for Family Coverage if she or he is: under eighteen years of age; over eighteen but a Full-time Student and receiving a majority of their support from the Insured, but not more than twenty-four years of age; or if he or she is Dependent due to Disability or Illness.*
>
> Your interest rate is the average prime rate (as published in the Wall Street Journal) for the thirty days preceding the issue date of your statement plus 4.5 percent.
>
> An Employee may have 0 or 1 Spouse
>
> Py = (1/360 * FA) + (1/12 * (.075 * CB))

The preceding are examples of business rules. Businesses are rife with these kinds of statements, formulas, conditionals and mandated relationships. Despite the ubiquity and importance of business rules, they are seldom accorded the status of first class objects. Instead rules are documented and presented as static relationships among classes. (Traditional development methods deal with rules in the same way, e.g. using entity relationships in a data model.) These static relationships are then supposed to be established and enforced by appropriate code in object methods or by establishing special purpose objects like a database management object that specialize in the maintenance of relationships.

Object thinking mandates that, "everything is an object." A rule should therefore be an object and, like any other object, have the ability to perform a specific set of services: modify itself and evaluate itself. By modify we mean the rule must be flexible, adding, deleting or extending clauses. By evaluation we mean applying computational rules to itself so as to return an object encapsulating a meaningful result – in the simplest case, returning a Boolean object that encapsulates whether the rule was satisfied or not.

Applying object thinking to the problem of creating a rule object leads to observations about structure and responsibilities.

Structural Abstraction of a SelfEvaluatingRule

A useful object thinking heuristic for decomposition is to take an occurrence of such an object as it appears in the real world and separate its physical parts. We can do this with a simple example of a rule expressed as a formula:

X = 4q + (p*r)

Analysis of this equation (rule) yields five distinct types of physical components:

- **X, q, p** and **r** represent things which are currently unknown but knowable. Following convention, we will call theses items **variables.**
- **4** represents a **constant** value.
- **+** and ***** are behavioral **operators.**
- **(** and **)** in combination represent an aggregation and precedence **operator.**

- = is an **operator** of symmetry - indicating that the thing on one side of the equation is in some sense symmetrical with, equal to, or to be assigned to the thing on the other side.

The order in which these components appear is important, as is the association of elements with each other (e.g. determination of the receiver, operator, argument relationship). Our initial, structural, abstraction of a rule then follows:

A rule is an ordered collection of variables, constants and operators.

Each of these components can be analyzed to discover its behavioral characteristics. Constants are perhaps the simplest, most often being instances of known classes like Integers, Float, Reals, Character, String, *et cetera*, although nothing prevents them from being any known object with a fixed or constant value.

Variables are more interesting. At first glance we seem to have two different types: those, like X, that equate (are assigned) to the resolution of the entire rule; and those, like **p**, **q** and **r**, that represent a discrete value which can be used to resolve the rule.

If we think of variables as a place where a value (an object) could be but currently is not, we can posit a structural abstraction for a variable. In the case of variables like **X**, that abstraction would consist of a "targetObject" and a "setterMessage". For the q, p and r variables, the abstraction is a "sourceObject" and a "getterMessage". Behaviorally, a variable is responsible for obtaining its value, for instantiating itself. It does so by sending the getterMessage to the indicated sourceObject.

In the realm of objects we are accustomed to treat operators as messages. In our simple example we see three nuances of operator: first, those representing standard messages sent to objects to invoke behavior (+ and * being examples); second, those that instruct the compiler/interpreter/virtual machine to establish precedence; and third, assignment operators. The hardest part of implementing self-evaluating rules was dealing with the relationship between operators and the objects related to each. The solution turned out to be creating a "term" class that had receiver, operator, and (optionally) arguments as instance variables. This turned our rule into an ordered collection of terms, but the conceptual essence of self-evaluating rules was preserved.

Our attention can now turn to behavior and the dynamics of the Rule and Term structures we have defined.

Behavioral Abstraction

An object cube for a self-evaluating rule would have the following responsibilities on side one: modify self and evaluate self. The second responsibility requires collaboration with the ruleElements, i.e. the constants, variables, and operators.

Side two might show a stereotype of ordered collection – reflecting the modify self responsibility coupled with the definition of a rule; "an ordered collection of variables, constants, and operators."

Side four would indicate a need to know about what ruleElements need to be added, deleted, etc., but, surprisingly, little else.

Side five would define messages for: add an element at a location; delete a designated element; evaluate, instantiate, and resolve. Side three would show that evaluate, add, and delete messages would be public while instantiate and resolve would be private.

All basic behavior is now defined. A rule is asked to **evaluate** itself. It, in turn, sends itself the instantiate message causing itself to iterate across its elements and ask each variable object to instantiate itself (turn itself into an actual value by sending its message to its receiver). Once all variables are instantiated the rule then sends itself the message to resolve – apply the operators appropriately and return the result (with assignment taking place as needed).

Rules are recursive. Any element of a rule can be replaced with a rule. More than enough flexibility and power to get yourself into trouble unless you use other object thinking principles (e.g. simplicity and local action) to eliminate the need for truly complex rules.

Every object can be given the responsibility to validate itself simply by giving it a collection of rules to use as collaborators in that process. Those rules can be modified at run-time, just like any other object, merely by sending appropriate modification messages. Variables can be context sensitive in their instantiation.

This type of self-evaluating rule was presumed in examples in previous chapters. The form example assumed that every entry field would have a collection of self-evaluating rules that it would use to validate its contents. For example:

```
RETURN True IF (objectEntered whatIsYourClass) EQUALS 'Integer'.

RETURN True IF  10 < (objectEntered value) <100.
```

The form would have a separate set of rules for "cross-field" validation. For example a rule that said something like,

```
    RETURN true IF (zipcodeDirectory COMTAINS
((widgetNamedZipcode yourValue) CONCATENATEDWITH
(widgetNamedCity yourValue))
```

Creating rules as first class objects is essential to eliminating the kind of 'control' and 'management' that is endemic to classical software where data is assumed to be passive and special purpose objects (like database management systems) assume responsibility for validating all of the datums they contain – with inordinate degrees of complexity as a byproduct.

Implementation

Having thought about all aspects of your objects, your stage, and your scripts you are ready to make the step to implementation, ready to write code – almost. If you are an XP developer you are ready to write tests. Whatever kind of developer, you need to think about two more details, what your tests and code might look like and some additional information that must be provided to those objects whose primary role is the maintenance of bits of knowledge ("data" objects).

Methods

For every message you listed on side five of the object cube, you have to write a method and a test of that method (or tests). The first tests are obvious, when the message is sent to the object is an object returned? Is it an instance of the right class? Is it in the correct state? By state, I mean does it

have a reasonable value; or, if it is a composite object, do all of its instance variables have reasonable values?

The next set of tests concern an object's ability to recover from errors that it might encounter between the time it is asked to do something and the time it returns an object in response to that request. These tests are trickier because they usually involve a group of objects and a script. A very simple case would be an error that occurs when an entryField object cannot validate itself with the entered object. It must then invoke a dialog box object, ask the dialog to display an appropriate error message and accept an alternate value, and then re-perform the validation tests.

We have talked a lot about scripts and noted the possibility of script objects actualizing the sequence of communications among a group of objects. In practice, a lot of the communications we talked about in terms of scripts will be implemented in methods. Thinking about scripts helps you design your next set of tests. Does the precondition implied in the script always cause the appropriate message to be sent to the appropriate object? Does that object implement the next bit of the script? Does that implementation satisfy the conditions necessary to continue with the larger script?

The filling in of a form is an example of scripts being implemented in methods rather than in an independent script object.

- The form will receive the instantiate message. The form's instantiate message will be two lines (in some languages) of code: "self, iterate across all entry fields and tell them to instantiate themselves;" and, "self, send yourself your validate message."

- Each entry field will receive the instantiate message. That method will be about three lines: "self, ask your source of input (an object in one of your instance variables) to display itself and return a value to you;" "self, ask your validation rules to evaluate using the entered value;" and, "self let the form (whoever sent you the instantiate message) that you have successfully completed your job."
- Each validation rule will receive the evaluate message. The evaluate method will have two lines: "self instantiate," and "self resolve."
- Then the validation rule receives the instantiate message. Instantiate consists of one line: "self, tell all your elements that are variables to instantiate themselves."
- Then the validation rule receives the resolve message. Resolve consists of one line: "send the message value to each of your terms in a recursive fashion."

Writing tests for each of these methods (script fragments) is relatively straightforward – until you start to deal with exception and error handling. But even this complication can be ameliorated, by creating a new script that might be implemented in several methods in other objects.

Your tests provide any additional specification you might need regarding the form of your methods. Your selected (or mandated) programming language provides you with the syntactic rules that must be adhered to in order to implement the semantics implied by your tests. Unfortunately, most programming languages were not designed specifically to make it easy to express the semantics that object thinking yields. You almost always will have to perform some translation.

Knowledge Maintenance Objects

A lot (perhaps most) of the objects in your application will have the primary charge of maintaining a bit of information. It is convenient to call these "data" objects – as long as you remember they are not passive bits of data, they are real objects!

In addition to the fact that it exists, each data object needs to know a lot of additional information, information that traditionally was specified in a data dictionary or a CRUD matrix.

> A CRUD matrix gets it name from the actions: Create, Read, Update, and Destroy. It is a simple two dimensional table that associates users and data items. Each datum in an application appears at the head of a row of the matrix, each user (user role) appears as a column header in the matrix. Each intersecting cell then contains one or more of the characters, "C, R, U, or D" reflecting that users privileges (accesses) to the datum.

Items of importance from a data dictionary include:

- The name of each datum.
- The composition of each datum – e.g. Name is-composed-of an optional Honorific, and a First Name, and an optional Middle Initial, and a Last Name, and an optional Generational.
- The allowed size or upper and lower size limits; e.g. a string used for a First Name might have to contain at least one character but no more than thirty-five.
- Any initial or default value.
- A range of values – what data modelers call a domain; e.g., a Zip Code datum would have a domain of all the values listed in the USPS Zip Code Directory.

It will be essential to obtain this information about all your "data" objects. But what do you do with it when obtained? Where is it implemented?

Earlier we suggested that certain classes are most likely to be used to maintain knowledge – for example, characters, numbers, strings, dates, times, and money – which we called "primitives." The

obvious place to store CRUD and data dictionary information is in the objects that use that information. However it is unlikely you will actually want to modify all the classes that can be used as primitives. Instead you might want to create a "Datum" class.

A Datum class is a way to reify passive data into a first class object. An instance of datum would have the following structure:

Label: a string that names the datum.

Class: the name of the class of which this datum is an instance.

Value: the actual value of the datum – this would be an instance of on of the "primitive' classes.

Composition: an optional collection of datum objects that actualizes the data dictionary definition.

Validation Rules: a collection of self evaluating rules that would enforce domain constraints.

Access Rules: a rule that enforces the line of the CRUD matrix pertinent to this particular datum.

If you elect the "Datum" class option you would then return to your Object Cube, side 4, and replace a lot of the "primitives" listed there as datum objects. Take care to replace only those "primitives" that are used more or less exclusively as knowledge holders. If you are using a string or number, for instance, for behavioral capabilities and not just to hold a value, then you will not want to replace them with datum objects.

[1] Roads connecting cities are obviously real. But the existence of a road does not mandate use by vehicles, traffic remains a potential not a certainty.

[2] Some of these registrations will be made as part of the application initialization process – getting the cast (objects) ready to perform and others will come and go during the running of the application.

[3] Many years ago I formulated the basic ideas of self-evaluating rules and lectured my classes on the concept. I did not actually have a rule object implemented (leaving that as an exercise for the student). Without implementation self-evaluating rules were a nice idea, but …. Kevin Johnson, one of my graduate students that later became a colleague and great friend, would occasionally complain that rules could not be implemented as I was describing them (as described in this chapter). So one day we sat down and implemented them – with him doing almost all of the programming – in Smalltalk. The final concept of self-evaluating rules is the result of Kevin and I discussing them over many months and finally implementing them in at least one object language. He an I co-authored an unpublished paper that is the foundation for most of what is said in this section. I want to thank Kevin and acknowledge his invaluable assistance in finalizing the ideas in this section.