8

Thinking Towards Design

Discovery involves applying object thinking to the problem of decomposition (finding the objects that reside in the domain) and the problem of requirements (what behaviors are expected of individual objects and aggregations of objects as they interact in that domain). Tangible results of discovery probably include partially completed object cubes and a set of stories. These results take tangible form because they have value as a kind of external memory for the group of individuals involved in development.

The CRC Card method of Beck and Cunningham moved directly from discovery into implementation. Once you had a set of cards and stories you started to write code, doing some design oriented thinking about the code as you went along. XP follows the CRC Card method closely – pick up a prioritized story card, grab a partner, and start writing code. XP does require you to write code to be used in testing your work before actually writing the code to implement your objects and stories. Design, in XP, is not a separate process, it is infused in the process of writing tests and code.

Moving directly to the coding process may work for those whose brains are already saturated with object thinking – those who have internalized the ideas and ideals to the point that object thinking is automatic and largely non-conscious. For those just learning object thinking (or XP) there is a potential pitfall – the metaphors and definitions that served you well in discovery cease to provide valuable guidance as you think about design and implementation. Even worse, the 'good" metaphors are replaced with "bad" metaphors, metaphors reflective of "computer thinking" as discussed in Chapter 1 – Object Thinking.

Thinking towards design suggests the continual application of object ideas and metaphors as the developer's attention shifts focus away from the domain being modeled to the realization of the simulations of that domain – to the design of object and story simulations and, subsequently, to writing the code that will embody those designs.

"Thinking" towards design implies that it is not necessary to actually construct formal, documented, designs – only that your actual activities continue to be guided by object ideas and metaphors. Tangible documents and models suggested in the rest of this chapter are intended to illustrate object thinking about design and implementation and provide a convenient means of sharing ideas among a group of developers. As you internalize object thinking, the use of tangible documentation becomes less critical while remaining useful as a kind of checklist you can use as a substitute for perfect memory (both individual and collective).

Object Internals

Discovery has provided a decomposition of a domain into objects, an assignment of responsibilities to those objects, and a description of how objects might communicate and cooperate with each other to complete tasks beyond the capabilities of individual objects. This is not, usually, sufficient information to actually build computer software simulations of those objects and of those interactions.

Some examples of why this is so:

- Side one and two of the object cube (and the classical CRC Card) only capture behavioral expectations of your objects – they tell you nothing about the internal construction of those objects and nothing about the means used by those objects to fulfill their responsibilities.
- Assignment of responsibilities to an object tells you nothing about how to invoke that responsibility and nothing about what form the response will take.
- Identifying a need for collaboration tells you nothing about how the client object uses its collaborator, nor anything about how it knows of the collaborator's existence and location, nor anything about the form of communication with that collaborator.

There is a clear need to think about how your objects will be simulated - to make decisions about construction (design). Those decisions must be guided by the principles of object thinking and based on the intrinsic needs of the object. *An intrinsic need being defined as the means for fulfilling an assigned responsibility*. It is necessary to discern additional information about our objects and their needs. It is inevitable that the process of determining this information will involve making decisions about implementation, normally an activity associated

with design. By focusing on the intrinsic needs of the object we preserve application independence - maintain verisimilitude with the domain.

Knowledge Required

If we think of an object as if it were a human being (anthropomorphism) and we give that object a task it is appropriate to think about what that object may need to know in order to complete its assignment. We can use our understanding of human beings and what they need to know to perform tasks as a metaphorical guide for our thinking about object knowledge.

For example: suppose we ask Sara to ride her bike to the store and bring home some milk. She will need to know:

- How to ride a bicycle.
- Where the store is located.
- A route between home and the store.
- Perhaps, if more than one store is nearby, which store we wish her to go to.
- The actual quantity of milk required – disambiguate the word, "some."
- If she is a teenager, when we want her to perform this service.

Actually the list of things required is potentially very large. A lot of what Sara needs to know, we assume she already does – like how to get up from the couch, how to walk to wherever her bicycle is located, etc. We have criteria (usually tacit) that limits our listing of what Sara

needs to know to a few details. We have similar criteria for considering what an object needs to know in order to fulfill its responsibilities.

If we ask Sara to ride her bike, we probably already know she has claimed that ability and we trust her in making that claim. If an object says it can identify itself, we assume (trust) that the object has a method – a block of computer code – that actually performs the identification service. We do not need to explicitly list such abilities as required knowledge. (We will explicitly name the methods themselves when we specify how an object's responsibilities are to be invoked, i.e. when we specify a message protocol.)

We will usually assume that Sara knows which store and the route to take *unless* she has more than one option *and* if it matters to us which option she selects.

We will also assume that "now means now" and that the service is to be performed immediately upon receipt of the request. Again, *unless* we want the option of requesting the service at a specified time or in a particular set of circumstances.

Most of the time, we are interested in recording the information – knowledge – required by the object in order to fulfill its advertised services. Information is perilously close to being what we typically think of as being "data." It is therefore very easy to fall into the "computer thinking" trap of assuming knowledge required = object data structure or object attributes. Object thinking will help us avoid this trap.

The responsibilities recorded on side one of the object cube drive our thinking about the knowledge required. Look at each responsibility and ask what will the object need to know in order to fulfill this task. List your answer – as a descriptive noun or noun phrase - on face four of the object cube. Figure 8.1 shows side one and four of the airplane object cube introduced in the 'Another Example' sidebar in Chapter 7 – Discovery.

F08xx01

Figure 8.1 – Airplane Object Cube

Sides one and four of the Airplane object cube showing the relationship between responsibilities and knowledge required.

How did object thinking lead to the results recorded on side four of the object cube for the airplane?

- The airplane is responsible for identifying itself. It therefore needs to know its id. We record the noun (actually an abbreviation), "id" on side 4. [See sidebar, 'Object Cube Idiom,' for additional explanation of why things are recorded the way they are in the examples.]
- It needs to report its current location – hence "currentLocation."
- It needs to move to a new location on request – therefore "newLocation."
- Because side one of the object cube indicates that serving a request for current location requires a collaboration with the airplane's instrument cluster it needs to know of the instrument cluster itself – so we record "instrumentCluster" as a piece of knowledge required.

In most cases the list of knowledge required will be fairly short and reasonably obvious. In some cases a single responsibility might yield more than one piece of required knowledge.

Completing a list of items to be known is but the first step in thinking about the object's knowledge requirements. We have two other decisions to make about each piece of knowledge recorded in our list: how it is obtained and what form it takes (what class will be used to encapsulate that knowledge).

Object Cube Idiom

A number of factors play a role in deciding the actual form – the actual words and symbols – used to record information on an object cube. Paramount among these is the need to be explicit and avoid ambiguity. For example: the expectations of an object should be obvious from the phrases selected to record those responsibilities; the name of a piece of knowledge should unambiguously describe the semantic understanding of that knowledge; and, the names given to classes and methods should reflect the essence of those classes and methods. (In this regard we are very Confucian in our insistence that "only if things are given the proper names will all be right under Heaven.")

Countering the need for explicitness is the need for brevity. Eventually most of the names will be used in writing program source code. No coder really likes to type long descriptive names.

Another influence: the syntax of the programming language that the development team is most familiar and comfortable with. Smalltalkers will be quite comfortable recording "id: aString" as a method name or using "Integer" as a class name; but, C++ programmers would be more likely to use "id(string)" and "int" in similar circumstances.

This author, like everyone else, is a victim of his past – the idiom I am most comfortable with derives from the use of the Smalltalk programming language and its associated style and idiom. Although I will try to be as non-language specific as possible in my illustrations, be forewarned that some idioms and conventions will inevitably creep in.

Another example of idiom – naming conventions like the one that suggests that the method names for retrieving the object in a named variable and for placing a object into that variable are the same as the variable name itself – with the addition of an argument in the case of the "put" method. Example: the airplane has a piece of knowledge named "id." A method for retrieving the object encapsulating that id (usually a string) would be simply, "id." The method name for replacing the id string with another string would be, "id(aString)."

The proper idiom for use on object cubes, in code, and any other phase of modeling or development should reflect the community doing the development. It is the responsibility of the developers in your organization, or your domain, to determine appropriate idiom and to train new members of your development community in the use of that idiom.

An object has four different ways to gain access to the knowledge it requires.

- It can store that knowledge in an instance variable.
- It can ask for that knowledge to be provided along with the request for service (the message).
- It can obtain it from a third party - another object.
- Or, it can "manufacture" the knowledge at the point when it is required.

Upon deciding which of these options is most appropriate we will record that decision by noting an appropriate symbol next to each piece of knowledge. The symbols used are arbitrary, but in the examples in this book we will use: (V) for variable, (A) for Argument, (C) for collaboration, and (M) for method. A couple of heuristics for deciding which option to use:

- Objects are lazy. Every time you decide to store a piece of required information in a variable, the object must assume responsibility for maintaining that variable – it must add the capability to retrieve and to update the contents of that variable upon request. So, whenever possible and appropriate, use argument (A) and method (M) instead of variable.
- Collaboration is a form of dependence. Deciding to use a collaborator to obtain required knowledge makes you dependent on that collaborator. Objects strive to independence and so collaboration should also be minimized to the extent possible.
- Remember the definition of collaboration – requiring the service of an object not found inside your own encapsulation barrier (i.e. objects stored in instance, class, or temporary variables and objects received as arguments to messages) – and recognize that collaboration requires direct (e.g., you know the actual name/id of the object used as collaborator) or indirect (e.g., you know where to find the object, in a global variable perhaps) coupling with that collaborator object. Coupling is just as undesirable in the object thinking as any other development method or approach.
- In addition to knowing who your collaborator is (e.g., the instrumentCluster piece of knowledge recorded on side four of the Airplane object cube example), you might also want to add a piece of knowledge for the message to be sent to that collaborator as another item in your list of knowledge required. More often than not, the message used to invoke the collaboration is hard coded in the method where the collaboration is actualized. Recording the message itself as a separate piece of knowledge adds design flexibility and has the effect

of ameliorating any changes in coding required when and if the collaborator changes its interface.

- You might want to add the symbol (G) as an option for recording how you are accessing a piece of knowledge. (G), would stand for collaboration with a global variable. Starting down this road, however, might lead to making a distinction between a global in an application (G_a), a system global - e.g. the system clock – (G_s), or some sort of intermediate – like the pool dictionaries in Smalltalk – (G_p). If making such distinctions helps you and your group, there is not harm in using them. Take care that your object cube does not reflect a particular implementation context to the point that the general utility of the object is lost.

A final decision about knowledge required is to determine the kind of object that will embody (encapsulate) the information. A class name will be recorded for each piece of information listed on side four of the object cube. (Using a class name rather than a program language type, is preferred for this purpose because some information is complex – i.e. not a simple primitive.)

In some cases you will discover an entirely new kind of object when making this decision. The location object, for example, used to encapsulate the various components that make up an airplane's location: altitude, latitude, longitude, and vector.

Discovery of the location object is a direct result of applying object thinking to the question of knowledge required. As you think about what a location really is you discover the various component values that make up a location. You apply the "lazy object" principle and find out that both the airplane and the instrument cluster find keeping track of the location components is too taxing and needs to be delegated to someone else. The other candidate is an

instrument, but no instrument should know or try to keep track of any value except the one specifically generated by that instrument. Hobson's choice – create a new object with responsibilities reflecting the recording and maintaining of the values comprising a location.

Figure 8.2 shows sides one and four for the objects in the ATC example from the sidebar in the previous chapter. Except for the location object – just discussed – the selection of encapsulating object is pretty much a matter of common sense coupled with a knowledge of the existing or planned class library for your domain.

F08xx02

Figure 8.2 – object cubes, ATC example
Sides one and four of the objects (classes) introduced in the ATC example – sidebar, previous chapter.

Figure 8.3 shows side one of the objects in the Mortgage Trust application discussed in the previous chapter and Figure 8.4 shows the knowledge required for those objects. Some specific points about side four illustrated by those objects include:

- One
- Two
- Three

F08xx03

Figure 8.3 – Object Cube Side One
Objects classes from the mortgage trust application introduced in the previous chapter.

F08xx04

Figure 8.4 – Object Cube Side Four
Knowledge required for objects and responsibilities identified for the mortgage trust application introduced in the previous chapter.

Message Protocol

Side one of the object cube tells us what an object can do but reveals nothing about how to ask for the advertised services. We know from object thinking in general that services are invoked by sending a message to the object providing the service, but what form must the message take? This is not a trivial question because the form of the message is arbitrary but it must be exact or the receiving object will ignore it. (Actually it will cause an error if the message is wrong – a variation of, "I haven't the foggiest notion what you are asking me to do.")

We also have no clue about the way the object will respond to any request sent its way. Will it provide us something in return? In many cases we hope so. If it does, what will be the nature of the returned item?

To answer these questions we use side five of the object cube to record a message protocol – a list of messages and their associated responses. As with knowledge required, we refer to side one to elicit the necessary list of messages. We also apply whatever idiom and convention for message syntax employed in our development environment. As individual messages are recorded, care must be taken to maintain consistency with decisions made elsewhere on the object cube – notably the names and encapsulating objects noted on side four (knowledge required).

Figure 8.5 shows side one (responsibilities) and side five (draft message protocol) for the objects in the ATC example. Figure 8.6 and 8.7 show, respectively, side one and side five for the objects in the mortgage trust example.

F08xx05

Figure 8.5 ATC example objects
Sides one and five of the objects in the ATC example introduced in previous chapter.

Completing side five is usually very straightforward. The following heuristics help with any nuances involved:

- A full message signature includes four elements: the name of the receiver, the message selector (the actual message name), arguments (if any, arguments are optional), and the nature of the object returned to the sender of the message. Example: *aCollection includes (anObjectSpecification) aCollection*. *ACollection* is the receiver; *includes* is the message selector; *(anObjectSpecification)* is the argument; and, *aCollection* is the object returned. Because we are recording the messages received by the object whose cube we are completing, we omit the name of that object when drafting the message protocol – only the last three elements of the message signature are recorded on side five.
- Messages must be descriptive of the nature of the service (behavior) being invoked. It should be possible for a "naive" user of your object to immediately discern what is likely to happen if the indicated message is sent to the object. This includes specification of arguments – it should be obvious what kind of object(s) is/are being passed as part of the message.
- Messages must be relatively terse (programmers will get tired of typing long messages) a requirement that is at odds with the descriptiveness requirement.
- Messages should suggest correct implementation syntax – in concert with the standards and conventions adopted in your development environment.

- If you have decided, on side four, to store an object encapsulating some bit of information in an object instance variable you must add two messages to your protocol – one to obtain the object in the variable (a getter message) and one to replace it (a setter message). Most conventions use the variable name as the message selector. For example, if the variable is named "id," then the getter message would also be "id" and the setter message would be id(aString). The argument will reflect whatever decision we made on side four as to the nature of the encapsulating object for that variable.

- Some messages are imperative commands, "don't bother giving me anything back, just do this!" In those cases no object is returned. A commonly encountered convention is to note "self" as the object returned. For those more familiar with C++ and Java it is perfectly appropriate to put the term, "void," in place of self. The meaning is equivalent.

F08xx06

Figure 8.6 – Object Cube side one
Message protocol for the objects in the mortgage trust application.

F08xx07

Figure 8.7 – Object Cube side five
Message protocol for objects in the mortgage trust application

Message Contracts

Side three of the object cube has not been forgotten – it is only now that it will make some sense to talk about what is recorded on that face of the object cube.

Contracts are actually a historical artifact – a concept introduced by Rebecca Wirfs-Brock and her co-authors in their book length treatment of the CRC Card method invented by Beck and

Cunningham. The idea was to aggregate responsibilities – later messages – into groups to reflect the users of those methods. This particular use of contracts did not gain wide acceptance and the idea of contracts became rather obscure.

A concept from programming – message scope or visibility – provided renewed use for contracts. In a programming language like Java, methods (and their invoking messages) could be designated public (anyone can send that message and invoke that service), private (only the object itself could send the message to itself), and protected (only a designated group of user objects could send the message). Other languages – most notably Smalltalk – did not make provision for such method/message declarations. Using contracts to at least specify the intent of the object developer as to the proper use of messages was a natural extension of the notion of contracts. If the implementation language supported message scooping, side three provides a specification to the programmer. If not, side three documents the intended use of the categorized messages – and no good object programmer would misuse private or protected messages. (Smile.)

As class libraries grew in size and the messages associated with individual classes grew in number (something that should not have happened if object thinking had guided the design of those classes) it proved useful to create sub-categories of classes and of methods simply to simply the search for an appropriate class or method in the library. A typical browser in an integrated development environment (IDE) might show for lists: Categories of classes, class list, categories of methods and method list. A user would select the desired class category and see

only those classes in that category displayed in the second list. Then select a method category and see only those methods included in that category displayed in the last list. Typical method categories include: accessing, displaying, updating, calculating, etc. Such categories can be captured on the object cube as contracts on side three.

The layout of side three is simple: A contract name followed by an indented list of the messages intended to be included in that contract. A message can appear in more than one contract unless the contracts reflect programming specification of public, protected, and private.

F08xx08

Figure 8.8 – Object Cube side three
Contracts for all classes in both the ATC and mortgage trust application examples.

State Change Notification

"Objects encapsulate state." When this claim is made it is usually a reference to changes in the objects occupying an instance variable. This idea reflects the common definition of state – a change in value of any aspect or characteristic of a thing – colored by the way state is used in data driven object design.

If an object is properly designed it should be so simple as to have very little interesting state. Some examples of state might include:

- An object in an instance variable has been replaced with another object – the object (or value of that object) matters far less than the fact that the change occurred so we would characterize the new state as "changed."
- An object might be in the process of responding to a message and therefore unavailable to receive a new message. State = "busy.
- An object might be "defective" in some sense, out of calibration or completely inoperative, yielding states of "faulty" and "dead." (In the latter case we are not looking for "dead" so much as, "I'm dying, gasp, gasp … the butler did it" written in blood with the object's last exhalation.)
- An object might be un-initialized, none of its instance variables contain objects other than Nil.
- From the world of persistence (databases) a variation on the "changed" state can be surmised – "dirty" which means a change that has yet to be reflected in the persistent persona of the object. There is an inconsistency in value between the cell of a database table and the object stored in an instance variable.

Although it is possible for an object to have numerous states, only a few of those states are likely to be of any interest to anyone outside of the object itself. In those cases where other objects might be interested in a state change, it is appropriate to list and describe those states on side six of the object cube. The syntax for side six is very simple – a descriptive name of the state and a short description of that state. Figure 8.9 shows side six for all objects in the ATC and mortgage trust examples.

F08xx09

Figure 8.9 – Object Cube side six
Events for objects in the ATC and Mortgage Trust examples.

Some important caveats concerning object state and side six of the object cube:

1. Side six records only those states that the object is willing to make visible to other objects. In this sense, side six is akin to side one in that it publishes part of the objects interface – the part of the object visible to others. Some state changes might be kept private to better reflect the domain being simulated by the object. For example, a selfCalibratingInstrument might have a state, "out of calibration," that it does not make visible to the outside world. Instead it detects that state itself, takes steps to correct that state, and only if it fails in such attempts will it generate a public state, "failed." Failed would appear on side six of the object cube, but "out of calibration" would not.

2. State changes are only visible to the object experiencing the change. Listing a state on side six does not imply you – or any other object – can see that change, only that you – and all other objects – may request to be notified when the object itself detects the change in itself.

3. Side six of the object cube does not capture, and is not intended to capture state related constraints on an objects behavior. That kind of information is captured in a static diagram – specifically a state chart – and will be discussed in Chapter 9 – All The World Is A Stage.

4. Advertising a willingness to notify others of state changes implies that the object has a mechanism for keeping track of who is to be notified, how, and for what. At first this implied requirement might seem to violate object thinking precepts – but it is quite

possible to satisfy such a requirement in a manner consistent with object thinking. The mechanism is an "eventDispatcher" object. Every object capable (and willing) of notifying others of its state changes must contain an eventDispatcher to effect that notification.

An eventDispatcher object can be visualized as a simple two-part table, as shown in Figure 8.10. The first column of the table contains events, one per row. The second column of the table contains a collection of eventRegistration objects. An eventRegistration, in its simplest form, is a tuple consisting of a *receiver* and a *message*.

F08xx10

Figure 8.10 – Event Dispatcher and Registration
An event dispatcher table and an event registration tuple

Creation of an eventDispatcher object increases flexibility by centralizing – but not controlling – the awareness of what events exist and who is to be notified when they occur. If an object decides to publish a new event a simple message to the eventDispatcher requesting the addition of a new row to the table is sufficient to effect that change. An object's eventDispatcher can be queried as to what events are available and it responds with a collection (a list) of the contents of the first column.

If objectA wants to know about a state change in objectB (one that objectB has advertised as public) it sends a registration of its own construction to objectB's eventDispatcher. ObjectA decides what message it wants sent to effect the event notification, which means objectA can

change its mind – use different messages in different contexts - simply by asking that its earlier registration be replaced with a new one containing a new message. This capability provides significant run-time flexibility, allowing changes without necessitating any changes in code and subsequent recompilation.

One other possibility to be noted: suppose we need to notify objects in a particular order, e.g. some objects need to be notified immediately and the needs of others are less urgent. We could change (probably subclass) eventRegistration to be a triple – receiverId, message, priority. We could then allow the collection that comprise the second column of the eventDispatcher to be a sorted collection. As eventRegistrations are added they are sorted according to their priority values.

Object Appearance

It would be difficult to talk about objects without talking about the graphical user interface (GUI). The Star project at Xerox PARC simultaneously advanced local area networking, GUI design, Smalltalk (object-oriented programming), and alternative input-output (I/O)modes (notably the mouse). Networks and I/O were sufficiently esoteric and close to hardware design that they went their separate ways, post-PARC. GUI design and objects, however, emerged so tightly coupled that it is often assumed that primary (if not exclusive) use of object design and object programming is the construction of graphical interfaces.

Visual development environments (like Visual Basic) managed to convey the impression that objects = the GUI widgets that appeared in the interface design toolbox. Once you had those widgets in place on your form object – you did real programming – sans objects. (Yes, it is true that the objects are still there, but they are not emphasized or enforced – with the net effect that most VB programs tend to be event-driven procedural in nature and not OO.) Even Smalltalk IDE tools like Parts (an extension for Digitalk Smalltalk) and VisualAge (IBM) emphasized the utility of objects for GUI building with far less emphasis on the objects behind the interface.

A worse error was propagated when many of the early tutorials on object programming introduced a misconception by suggesting a "method" colloquially referred to as "cocktail napkin design." It was suggested that the correct way to design an application was from the interface-in; e.g., sketch your interface on a cocktail napkin, then find the objects necessary to implement that interface. The GUI became a straight-jacket to which objects had to conform. This, in effect, meant that object design and implementation was little more than hard-coded reflection of the specific and idiosyncratic design of a set of visual interfaces. Change the visual interface and you had to change the object. Not only did this make for a lot more work, it meant that objects were not re-usable in different contexts if those contexts defined visual interfaces in an alternate fashion.

Object thinking acknowledges the special relationship between an object, "X," and some group of other objects whose role is to represent – visually or otherwise – object "X." At the

same time it suggest some important differences in the way that relationship is discovered and implemented.

Occasions Requiring an Appearance

While engaged in the task of discovering objects and responsibilities, we told stories about object interactions. Among the interactions in those scenarios were instances of an object displaying itself to a client. An example might be a character displaying itself on some medium for the benefit of an observing human being. What actually appears on the medium is not the character object; it is a representation of some aspect of that object (its visual appearance but not its behavior, or dynamics, or "soul"), just as your photograph is not you but a representation of some part (the surface part) of you.

The converse of displaying oneself to a client is to display the void at the core of your essence when you have yet to be instantiated. This time you display the void as a kind of request for service – asking a human user to please provide substance (a value) that will make you a fully instantiated object. A hybrid of both display and request is a display of your current value with the implicit request to change that value if appropriate.

Even the simplest object might have multiple representations of itself – just as you probably have more than one photograph of yourself. A character, for example, might have a 'bold' appearance and an italicized appearance. In cases where an object has multiple

appearances – and can be asked to switch from one to the other – it is necessary to add messages to the protocol that make this ability evident. For example, a date object might have two messages in its protocol: displayUS and displayEurope; the difference between the resulting display being a transposition of the day and month values.

Most objects are compositions of simpler objects – remember that every instance variable an object might have contains another object. If an object with instance variables has a gestalt (view of the whole) appearance, that gestalt would necessarily be a composite of the appearances of itself and all of its contained objects.

It is easier to illustrate ideas about an object having multiple appearances using visual metaphors. But visual representations are not the only appearances an object might have. Imagine an object that needs to store itself in a relational database. The RDBMS cannot accept the object in its natural glory, so the object must marshal[1] itself into a stream of bits that can be accepted and held by the RDBMS. The resultant stream of bits constitutes an appearance of the object, but is not the object itself, any more than is a visual representation.

Glyphs

The application of object thinking to the issue of text characters, numbers, and graphical symbols should lead to the recognition that all of these are really just instances of a single kind of

thing – a glyph. A glyph object would have one major responsibility – to display itself. In order to fulfill that responsibility it would need to know:

- Its origin – a point – used in the definition of its extent and in placement of its value on a medium.
- Its extent – an area (not necessarily a regular area, but likely some sort of polygon).
- Its scaling factor – analog of point size, some constant that would allow it to occupy greater or lesser extent.
- Its orientation – a radian (used if the glyph is to be laid out other than horizontally).
- Its value – something as primitive as a bitmap or, more likely, an algorithm (vector) that results in the generation of colored pixels on a medium that is the actual appearance of the glyph.
- Its ascii value – a bit stream.
- Its ebcdic value – a bit stream.
- Its Unicode value – a bit stream.

The message protocol for a glyph would include the display message (the glyph would use values in all the appropriate variables to create the bit stream sent to a printer or graphics card that actually effects the display); and getter and setter messages for each of the instance variables listed in the bullet list.

If glyphs existed in typical information system applications it would not have been a issue when a certain rock star changed his name to a Celtic-Egyptian symbol.

The separation of objects and views or appearances of objects is essential. It has also been addressed – but incompletely - in a number of different ways; in programming languages and architectural patterns (e.g. model-view-controller, presentation-abstraction-controller).

Object thinking would suggest that each and every object have a collection of appearances, would advertise (as a responsibility / service / message) its ability to appear in different guises as well as variations (e.g. point size) of a single guise. Each object would transfer to its appearance object (e.g. a glyph as described in the sidebar, Glyphs) responsibility for acquiring and maintaining the information required to effect each appearance.

Elementary objects, like characters, might have a small set of appearances each of which is a simple glyph. The appearance of more complicated objects, like strings and dates and numbers, would be composites – an ordered set of appearances of each of its constituent parts – each character in a string, each number and symbol in a date, each integer and symbol in a number. Still more complicated objects like a form would be a composite of its own appearance (a boundary perhaps) plus the composites of each of the elements appearing on the form plus the composites and glyphs of each member of each element that appears on the form.

Keeping track of this apparent complexity is relatively straightforward – you just apply the composite pattern. Composite is one of the twenty-six patterns in the first pattern book – Design Patterns. Applied to appearances – the pattern asserts that every appearance is made up of glyphs or appearances. A glyph being a leaf node in the hierarchy and an appearance being an instance of appearance which in turn is made up of appearances and/or glyphs. To display an

appearance, you use recursion to traverse the hierarchy display all glyphs and decomposing all appearances into glyphs and appearances until you reach a level containing nothing but glyphs.

Widgets, Forms, Reports

Visual programming languages typically provide a number of different widget types to be used in collecting and displaying bits of information. A character widget, a number widget, a date widget, a currency widget, etc. are examples. Object thinking suggests that there is a need for only one type (class) of data entry widget. A dataEntryWidget would have an appearance, probably just a rectangle or a bit of underlining. It would have behavior that included signaling whenever it had been changed (a person entering a value in the box or on the line). It would also possess a set of rule objects that it could use to validate its own contents. An example of a rule might be: valueEntered is a date, if true, return True else return False. Another example: valueEntered is between minValue and MaxValue. Because the rules are objects and because each widget has a collection of rules that it can apply to itself – a collection that can be modified at any time with add, delete, replace messages – creating any type of specialized widget is trivial. (See discussion of self-evaluating rules in Chapter 10 – Issues and Examples.)

A Form, in light of object thinking, is nothing more than an ordered collection of elements. An element might be a string or a dataEntryWidget. A form would have responsibilities to: display itself – collaborating with its elements; update itself (add and delete elements) to effect different instances of itself: fill itself in – collaborating with its

dataEntryWidgets; and, validate itself - working with its own collection of rule objects. Form validation rules exist for assuring consistency among the values stored in dataEntryWidgets (each of those widgets having already validated their contents according to their own set of rules). For example, making sure that the value stored in the Zip Code widget is consistent with the value stored in the State widget.

A report is similar to a form – a collection of elements. The form adds and deletes elements to become different instances of a Form. The form displays itself by asking each element to instantiate itself – obtain a value for itself – and display itself. Each element has its own set of appearance rules, which it uses to display itself. An example of an appearance rule is, "my point size is 1.5 times the default point size for this report."

Both reports and forms make use of the glyph and composite patterns discussed earlier in this section.

Object State, Object Constraints

Readers familiar with typical treatment of object development might be curious as to the absence, so far, of any discussion of object state other than the willingness to notify others of state changes as recorded on side six of the object cube. Some definitions of objects suggest that

they "encapsulate state" and modeling methods and tools, like UML, provide sophisticated models for capturing state-of-the-object information.

Object thinking shows little concern about state – when discussing objects specifically because a properly designed object has very little interesting state. What state it might have should be private – behind the encapsulation barrier – except to the extent the object is willing to make public the fact that a state change occurred.

Most discussion about object state are really about state based constraints to be imposed on an object. Such constraints are not intrinsic to the object itself, they are an aspect of the situation in which the object finds itself employed. This kind of state modeling is important, but not important in advancing our understanding of individual objects. For this reason discussion of state modeling involving objects will be taken up in Chapter Nine, All the World's a Stage.

All other constraints that might be imposed on an object are also reflective of a situation, not in the object *per se*. Table manners, for example, reflect a set of rules that promote or inhibit intrinsic human eating behavior – fingers and communal bowl in an Ethiopian ethnic restaurant, a plethora of special purpose utensils and prescribed behaviors at a formal state dinner. It is a mistake to attempt to incorporate this kind of variation in an individual object's specification. This type of rule-based constraint will also be addressed in Chapter 9.

[1] Marshalling involves the object taking itself apart, asking each part to convert itself into bits, and then asking each part to line up in some kind of order. The process is analogous to a modem converting an analog signal to digital (and eventually back again).