3

From Philosophy to Culture

Philosophy provides roots, determines some values and affects the design of some tools, but the fullness of the object thinking difference must be understood in terms of a broader context – as a culture. The cultural metaphor suggests the need to look for the 'shared, socially learned, knowledge (norms, values, world-views), and patterns of behavior (individual actions and organizational relationships) that characterize a group of people. (The philosophy discussed in the previous chapter is part of the cultural knowledge shared by object thinkers as members of an object culture.).

Robert Glass used the metaphor of culture to explain differences in how different groups of people conceive of and develop software and how the results of their work are evaluated. His contrast of "Roman" and "Greek" cultures directly parallels the contrast made in the previous chapter between formalist and hermeneutic philosophies. The 'Greek' culture described by class is a close match to the XP culture[1] and the object culture we will explore in this chapter.

Greek and Roman

Glass makes the argument for two cultures within the realm of software development, two cultures that frequently find them selves in conflict based on cultural values. He uses an analogy with Greek and Roman culture to illustrate the differences as follows:

> *In ancient Greece, an individual would act as his own agent in his own behalf, or combine with other people to act together as a team. In a Greek work environment, you bring your tools to work with you, you do your stuff, and then you pack up your tools and take them home. You are an individual – an independent contractor. You are not owned body and mind. You are merely providing a service for compensation.*
>
> *In Rome, one's first duty was to the group, clan, class, or faction upon which one depended for status. Known as gravitas, this meant sacrificing oneself for the good of the organization, and giving up ones individuality and identifying closely with the group. In a Roman environment you go to work, the company hands you your tools, and then it holds you and your mind hostage until you sever your relationship with the organization. You are not an individual: you are owned by the organization body and mind, twenty-four hours a day. There are substantial rewards for this however. The organization provides you with security, money, and power.*
>
> *Robert L. Glass*

Glass is particularly interested in the degree to which the two cultures support creativity and asserts that the Roman culture is likely to take the creativity, passion, and magic out of the work of software development. He further notes that Roman culture will emphasize up-front planning, control, formal procedures as a means of control, maximum documentation, and will value logical, analytical, thinking above empirical and inductive thinking.

Even a cursory evaluation of XP values and practices reveals their incompatibility with Roman thinking. When objects were first introduced, they too reflected a Greek and not a Roman culture. Smalltalk was motivated by a need to empower people, to make interaction with a computer "fun and creative." Exploratory development, a kind of rapid prototyping, was seen as the proper way to develop new software – as opposed to the notion of up-front detailed design and rote implementation favored by the (Roman) structured development culture.

Our exploration of the object culture begins with a rough enumeration of some groups likely to be included in this culture. The original proponents of object ideas (the original Simula team, the Smalltalk team), advocates of behavior-based object methods, the first XP and Agile practitioners are likely candidates, and, of course, any who recognizes themselves as a member of the 'Greek' culture described by Glass.

Absent a full ethnography, a cursory look at this culture reveals some important traits and characteristics. Specifically:

- A commitment to disciplined informality rather than defined formality.
- Advocacy of a local rather than global focus.
- Production of minimum rather than maximum levels of design and process documentation.
- Democratic rather than imperial management style.

· Commitment to design based on coordination and cooperation rather than control
· Practitioners of rapid prototyping instead of structured development
· Value the creative over the scientific.
· Driven by internal processes instead of external procedures.

Cultures will usually have an origin myth, various heroes and heroines, stories about great deeds and important artifacts, as well as a set of core beliefs and values. All of these are evident in the object culture and someday will be captured in a definitive ethnography. It is not my intent to elaborate that culture here, merely to draw the reader's attention to the fact that such a culture exists.

Being aware of object culture is valuable for object thinkers in three ways. First, it provides insight into the dynamics of interaction (or lack of it) between objectivists and traditionalists. Often the only way to understand the mutual miscommunications and the emotional antipathy of the two groups comes from understanding the underlying cultural conflict.

Second, and most importantly, it reminds the aspiring object thinker that they are engaged in a process of enculturation; a much more involved endeavor than learning a few new ideas and adopting a couple of alternative practices. Most of the material in the remainder of this book cannot be fully understood without relating it to object culture in all its aspects.

Third, it suggests a way to know you have mastered object thinking. When all of your actions, in familiar and in novel circumstances, reflect "the right thing" without the intervention of conscious thought; then you are an object thinker.

Subsequent chapters will deal with object thinking specifics, all of which are intimately related to a set of 'first principles' or presuppositions reflective of the object culture as a whole. The four principles introduced here are frequently stated, stated in a manner that implies the value of the principle is obvious. Just as a member of any culture makes assertions that are indeed obvious – to any other member of that culture.

Four Presuppositions

One: Everything is an object.

This assertion has two important aspects. One is essentially a claim to the effect that the object concept has a kind of primal status – a single criteria against which everything else is measured. Another way of looking at this claim would be think as object as the equivalent of the quanta from which the universe is constructed. The implication of this aspect of everything-is-an-object suggests that any decomposition, however complicated the domain, will result in the identification of a relatively few kinds of objects and only objects.

There will be nothing "left over" that is not an object. For example:

- "Relationships" which are traditionally conceived as an association among objects that are modeled and implemented in a different way than an object – would themselves become just another kind of object, with their own responsibilities and capabilities.
- "Data" which, traditionally, is seen as a kind of "passive something" fundamentally different from "active and animated things" like procedures. Something as simple as the character "D" is an object

– not a datum – that exhibits behavior just as does any other object. Whatever manipulations and transformations required of an object, even a character, are realized by that object itself instead of some other kind of thing (a procedure) acting upon that object. The common sense notion of data is preserved because some objects have as their primary, but not exclusive, responsibility the need to represent to human observers some bit of information.

- "Procedures" as a separate kind of thing are also subsumed as ordinary objects. We can think of two kinds of procedure – a script that allows a group of objects to interact in a prescribed fashion and the "vital force" that actually animates the object and enables it to exhibit its behaviors. A script is nothing more than an organized collection of messages and both the collection and the message are nothing more than ordinary objects. The vital force is nothing more than a flow of electrons through a set of circuits – something that is arguably apart from the conceptual understanding of an object just as the "soul" is deemed to be different from but essential to the animation of a human being.

Equating, even metaphorically, a "procedure" to a "soul" will strike most readers as a bit absurd but there is a good reason for the dramatic overstatement. It sometimes takes a shock or an absurdity to provide a mental pause of sufficient length that a new idea can penetrate old thinking habits. This is especially true when it comes to thinking about programming where the metaphysical reality of two distinct things – data and procedures – is so ingrained it is difficult to transcend. So difficult, in fact, that most of those attempting object development fail to recognize the degree to which they continue to apply old thinking in new contexts.

Take object programming for example – using Smalltalk as an example merely because it claims to be a pure object language. Tutorials from Digitalk's Smalltalk manuals illustrate how programmers perpetuate the notion that some things "do" and others are "done to."

The code in **Listing One - Pascal** is a Pascal program to count unique occurrences of letters in a string entered by a user via a simple dialog box. (Pascal was designed to teach and enforce the algorithms (active procedures) plus (passive) data structures = program mode of thinking.)

Listing Two – Naïve Smalltalk shows an equivalent Smalltalk program as it might be written by a novice still steeped in the algorithms + data structures mode of programming. Both programs contain examples of explicit control and overt looping constructs. The Pascal program also has typed variables – an implicit nod to the need for control over the potential corruption of innocent passive data.

Listing One – Pascal

```
program frequency;
      const
        size 80;
      var
        s: string[size];
        i: integer;
        c: character;
        f: array[1..26] of integer;
        k: integer;
     begin
        writeln('enter line');
        readln(s);
        for i := 1 to 26 do f[i] := 0;
        for i :=  1 to size do
            begin
            c := asLowerCase(s[i]);
            if isLetter(c) then
                begin
                k := ord(c) - ord('a') + 1;
```

```
                  f[k] := f[k] + 1
                  end
             end;
          for i := 1 to 26 do
             write(f[i], ' ')
       end.
```

There are some initial evidence of object thinking in listing two – mostly ones enforced by the syntax of the Smalltalk language – like the use of the Prompter object, the control loops initiated by integer objects receiving messages, the size of the string being manipulated is discovered by asking the string for its size etc.

Listing Two – Naïve Smalltalk

```
| s c f k |
f := Array new: 26.
s := Prompter prompt: 'enter line' default: ' '.
1 to: 26 do: [:i | f at: i put: 0].
1 to: s size do: [
      :I | c := (s at: i) asLowerCase.
      c isLetter ifTrue: [
         k := c asciiValue - $a asciiValue + 1.
         f at: k put: (f at: k) + 1.
                          ].
                  ].
 ^ f
```

A programmer better versed in object thinking (and, of course the class library included in the Smalltalk programming environment) starts to utilize the innate abilities of objects, including 'data' objects (the string entered by the user and character objects) resulting in a program significantly reduced in size and complexity as illustrated in **Listing Three – Appropriate Smalltalk**.

Listing Three – Appropriate Smalltalk

```
| s f |
s := Prompter prompt: ‘ enter line ‘ default: ‘ ‘.
f := Bag new.
s do: [ :c | c isLetter ifTrue: [f add: c asLowerCase]].
^ f.
```

Types, as implied earlier, create a different kind of thing than an object. Types are similar to classes in one sense but classes are also objects and types are not. This distinction is most evident when creating variables. If variables are typed they are no longer just a place where an object resides. Typing a variable is a non-object way to prevent all but a certain kind of object from taking up residence in a named location. Many people have advanced arguments in favor of typing but none of those arguments directly challenge the everything-is-an-object premise. Instead they reflect issues of convenience or accuracy for the software developer who is assumed to be very prone to errors in his or her coding.

The "everything is an object principle" applies to the world, the problem domain, just as it applies to design and programming. David Taylor[2] and Ivar Jacobson[3] use objects as an appropriate design element for engineering, or reengineering, businesses and organizations. (See sidebar, “David A. Taylor and Convergent Engineering.”)

David A. Taylor and Convergent Engineering

Traditional modeling of businesses and organizations are flawed, according to Taylor, because of the lack of consistency among the set of models utilized. For example – neither a financial model nor a data model captures the cost of a bit of information and inconsistency in design philosophy prevents the two models from collectively revealing such costs – they cannot be coordinated.

In his book, *Business Engineering with Object Technology* (1995, John Wiley and Sons), suggests creating a single object-model incorporating everything necessary to produce traditional financial, simulation, process, data, and workflow models as “views” of the unifying object model. His process for accomplishing this goal is “convergent engineering” and it, in turn, is based on a behavioral, CRC card, approach to object discovery and specification.

In addition to describing how to conceptualize objects and classes, he describes a process for discovery and specification leading to the creation of the organizational object model. That model identifies all the objects in an organization and how they interact – not just the ones that will eventually be implemented as software. He also provides a framework for business objects that illustrates the power of object thinking in generating simple but powerful objects. His framework defines four classes (Business Elements, Organizations, Processes, and Resources), describes the behaviors of each, how those behaviors contribute to the generation of the five standard types of business model, how they can be customized, and how their inter-operation can be optimized so as to reengineer the organization as a whole.

The programming example shown earlier illustrates one dimension of treating everything as an object. Applying the “everything is an object” principle to the world – finding and specifying objects that are not going to be implemented in program code or software – can be illustrated by considering a Human object. Objects, as we will discuss in detail later, are defined in terms of their behaviors. A behavior can be thought of as a service to be provided to other objects upon request.

What services do humans provide other objects? For many this is a surprising question because human beings are not “implemented” by developers and therefore considered outside the scope of the system. But it is a fair question and should result in a list of responsibilities similar to the following:

- Provides information.
- Indicates a decision.
- Provides confirmation.
- Makes a selection.

The utility of having a *Human* object becomes evident in the simplification of interface designs. Acknowledging the existence of *Human* objects allows the user interface to reflect the needs of software objects for services from human objects. This simple change of perspective – arising from application of the everything-is-an-object principle – can simplify the design of other objects typically used in user-interface construction.

Additional implications of the everything-is-an-object premise will be seen throughout the remainder of the book.

Two: Decomposition – the discovery and specification of objects - should be driven by an understanding of the domain and based on behavior as the differentiation criteria.

Four threads support the first part of this presupposition, i.e. domain driven decomposition:

- The system description language philosophy behind Simula as discussed above.
- Alan Kay's ideas about user illusions and objects as reflections of expectations based in an understanding of how objects behave in a domain.
- David Parnas' arguments in favor of "design decision hiding" approach to decomposition – partitioning the problem space and not the solution space as did functional decomposition approaches – as discussed above.
- Christopher Alexander's[4] ideas about design as the resolution of forces in a problem space and his subsequent work on patterns that underlay the organization of a problem space and provide insights into good design. These will be elaborated later in the book in the discussion of patterns and pattern languages as an aspect of object thinking.

The second part of the presupposition, using behavior as the criteria for decomposition requires some additional discussion, beginning with some reflection on the importance of decomposition in the overall design process.

Proper decomposition has been seen as ***the*** critical factor in design from very early times. A quotation from Plato is illustrative.

> *[First,] perceiving and bringing together under one Idea the scattered particulars, so that one makes clear the thing which he wishes to do... [Second,] the separation of the Idea into classes, by dividing it where the natural joints are, and not trying to break any part, after the manner of as a bad carver... I love these processes of division and bringing together, and if I think any other man is able to see things that can naturally be collected into one and divided into many, him I will follow as if he were as a god.*
>
> *- Plato*

Plato suggests three things: 1) decomposition is hard (and anyone really good at it deserves adoration); 2) any decomposition that does not lead to the discovery of things that can be recombined – composed – is counterproductive; and, 3) the separation of one thing into two should occur at "natural joints." By implication, if you decompose along natural joints – and only if you do so – you end up with objects that can be recombined into other structures. Also by implication, the natural joints occur in the domain and "bad carving" results if you attempt use the wrong "knife," – the wrong decomposition criteria.

If you have the right knife and are skilled in its use – know how to think about objects and about decomposition – you will complete your decomposition tasks in a manner analogous to the Taoist butcher:

> *The Taoist butcher used but as a single knife, without the need to sharpen it, during his entire career of many years. When asked how he accomplished this feat, he paused, then answered, "I simply cut where the meat isn't.*

According to this traditional story, even meat has natural disjunctions that can be discerned by the trained eye. Of course a Taoist butcher is like the Zen master that can slice a moving fly in half with a judicious and elegant flick of a long sword. Attaining great skill at decomposition will require training and good thinking habits. It will also require the correct knife.

Decomposition is accomplished by applying abstraction - the "knife" used to carve our domain into discrete objects. Abstraction requires selecting and focusing on a particular aspect of a complex thing. Variations in that aspect are then used as the criteria for differentiation of one thing from another. Traditional computer scientists and software engineers have used data (attributes) or functions (algorithms) to decompose complex domains into modules that could be combined to create software applications. This parallels Djikstra's notion that, "a computer program equals data structures plus algorithms."

The fact that a computer program consists of data and functions does not mean that the non-software world is so composed. Using either data or function as our abstraction knife is exactly the imposition of artificial criteria on the real world – with the predictable result of "bad carving." Neither the use of 'data' nor 'function' as your decomposition abstraction leads to the discovery of natural joints. David Parnas pointed this out in his famous paper, "On Decomposition." Parnas, like Plato, suggests that you should decompose a complex thing along naturally occurring lines; what Parnas calls "design decisions."

Both ‘data’ and ‘function’ are poor choices for being a decomposition tool. Parnas provided several reasons for rejecting ‘function.’ Among them:

- Resulting program code would be complicated, far more so than necessary or desirable.
- Complex code is difficult to understand and test.
- Resulting code would be ‘brittle’ and hard to modify when requirements changed.
- Resulting modules would lack composability – they would not be reusable outside the context in which they were conceived and designed.

Parnas’ predictions have consistently been demonstrated as the industry blithely ignored his advice and used functional decomposition as the primary tool in program and system design for thirty (forty if you recognize the most object development also uses functionality as an implicit decomposition criteria) years. Using data as the decomposition abstraction leads to a different set of problems. Primary among these is complexity arising from the explosion in total data entities required to model a given domain and the immense costs incurred when the data model requires modification.

Some examples of the kind of explosion referred to in the previous paragraph from my own consulting practice. One organization designed a customer support system that identified 15 different customer classes – because they were using a data oriented approach and had to create new classes when one type of customer did not share attributes of the other types. In a much larger example – a company had just completed a corporate data model (costing in the millions of dollars) when they decided to build a very large object system. They mandated the use of the data model for identifying objects – resulting in a class library of over 5,000 classes. This became the foundation of their system, causing enormous implementation problems. A final, mid-range, example was a database application for billing and invoicing where management demanded 1-1 replication of the existing system. This was accomplished, but it took over a year with an off-shore development team of 10-15 developers. My colleague and I duplicated the capabilities of the system, using object thinking in a weekend. Management, however, was not impressed.

What criteria should be used instead of data or functions? Behavior!

Coad and Yourdon claimed that people have natural modes of thought. Citing the *Encyclopedia Britannica,* they talk about three pervasive human methods of organization that guide their understanding of the phenomenological world: differentiation, classification, and composition. Taking advantage of those "natural" ways of thinking should, according to them, lead to better decomposition.

Behind the Quotes

Ed Yourdon and Peter Coad

Edward Yourdon is almost ubiquitous in the world of software development – publishing, consulting and lecturing for decades on topics ranging from structure development to various kinds of crises (e.g. the demise of the American programmer, Y2K).

The foundation for his reputation arose from his popularization of "structured" approaches to analysis and design. His textbook on structured analysis and design was a standard text through several editions. In 1991 he published two books, a new edition of *Structured Analysis and Design* and a small book, co-authored with Peter Coad, called, *Object Oriented Analysis*.

In the object book, Yourdon made a surprising admission – the multiple model (data in the form of an Entity Relation Diagram, process flow in the form of a Data Flow Diagram, and implementation in the form of a Program Structure Chart) approach advocated in his structured development writings (including the one simultaneously published) never, in his entire professional career, worked! In practice it was impossible to reconcile the conceptual differences incorporated in each type of model.

Objects, he believed, would provide the means for integrating the multiple models of structured development into one. Unfortunately, he chose data as the “knife” to be used for object decomposition. Other ideas advanced in that book proved to be more useful for understanding objects and object thinking – especially the discussion of natural modes of thought.

Peter Coad parted ways with Yourdon after the publication of this book and developed a method and an approach to object modeling and development that was far more behavioral in its orientation. He has several books on object development that are worthy of a place in every object professional’s library.

Classification is the process of finding similarities in a number of things and creating a label to represent the group. This provides a communication and thinking shortcut, avoiding the need to constantly enumerate the individual things and simply speak or think of the group. Six different tubular, yellow, and edible things become “bananas,” while five globular, red, edible things become “apples.” The process of classification can continue as we note that both apples and bananas have a degree of commonality that allows us to lump them into an aggregate called “fruit.” In doing so we create a taxonomy that can eventually encompass nearly everything - the Linnean taxonomy (and its more sophisticated DNA based successors) of living things being one commonly known example.

Composition is simply the recognition that some complicated things consist of simpler things. Ideally, both the complicated things and the simple things they are composed of have been identified and classified. Booch's 'Canonical Form of Complex Systems' captures both classification and composition hierarchies and the relationship that should exist between the two.

Classification requires differentiation, some grounds for deciding that one thing is different from another. The differentiation grounds should reflect natural ways of thought, as do classification and composition. So, how *do* we differentiate things in the natural world?

Consider a tabby and a tiger. What differentiates a tiger from a tabby, why do we have separate names for them? Because one is likely to do us harm if given the chance and the other provides companionship (albeit somewhat fickle). Each has at least one "expected behavior" that differentiates it from the other. It is this behavior that causes us to make the distinction.

Some (people that still believe in data for example) would argue that tabbies and tigers are differentiated because they have different attributes. But this is not really the case. Both have: eye color, number of feet, tail length, body markings, etc. The values of those attributes are quite different – especially length of claw and body weight but the attribute set remains relatively constant.

George Lakoff has written extensively on the issue of classification. His prototype theory is a close parallel to behavior as advocated in the object culture. His work is also a direct challenge to traditional attribute and set theory based ideas about classification and is controversial for the same reasons object ideas have been controversial.

Behavior is the key to finding the natural joints in the real world. This means, fortunately, that most of our work has already been done for us. Software developers simply must listen to domain experts. If the domain expert has at hand a name (noun) for something then there is as a good chance that that something is a viable, naturally carved, object.

Using behavior (instead of data or function) as our decomposition criteria mandates the deferral of much of what we know about writing software and almost everything we learned to become experts in traditional (structured) analysis and design. That knowledge will be useful eventually, but at the outset it is at best as a distraction from what we need to accomplish. We must re-learn how to look at a domain of interest from the perspective of a denizen (user) of that domain. We need to discover what objects she sees, how she perceives them, what she expects of them, and how she expects to interact with them. Only when we are confident that our understanding of the domain (and of its decomposition into objects) mirrors that of the user and the natural structure of that domain should we begin to worry about how we are going to employ that understanding to create software artifacts.

Three: Objects must be composable

As Plato noted, putting things together again is just as important as taking them apart. In fact, it is the measure of how well you took them apart. Any child with a screwdriver and a hammer can take things apart. Unless another child can look at the pieces and determine how to put them together again (or, even more importantly, see how to take as a piece from one pile and use to replace as a piece missing from another pile) the first child's decomposition was flawed.

Composability incorporates both the notion of re-usability and flexibility and therefore implies that a number of requirements be met:

- The purpose and capabilities of the object are clearly stated (from the perspective of the domain and potential users of the object) and my decision as to whether or not the object will suit my purposes should be based entirely on that statement.
- Language common to the domain, (i.e. accounting, inventory, machine-control, etc.) will be used to describe the object's capabilities.
- The capabilities of an object do not vary as a function of the context in which it is used. Objects are defined at the level of the domain. This does not eliminate the possible need for objects that are specialized to a given context, merely restricts redefinition of that same object when it is moved to a different context. Objects that are useful in only one context will necessarily be created but should be labeled appropriately.
- When taxonomies of objects are created it is assumed that objects lower in the taxonomy are specialized extensions of those above them. Specialization-by-extension means that objects lower in the taxonomy can be substituted for those above them in the same line of descent. Specialization-by-constraint (i.e. overrides) may sometimes be required but almost inevitably results in a "bad" object because it is now impossible to tell if that object is useful without looking beyond what it says it can do to an investigation of how it *does* what it says it can do. The exception – when a method is declared high in the hierarchy with the explicit intent that ALL subclasses provide their own unique implementation of that method and where the details of *how* are idiosyncratic but irrelevant from the perspective of a user of that object.

Although relatively simple to state, these requirements are difficult to satisfy. The general principle guiding the creation of composable objects is to discover and generalize the expected behavior of an object before giving any consideration to what lies behind that behavior. This is a concept that has been a truism in computer science almost from its inception. The most pragmatic consequence of this principle is the need to defer detailed design until we have a sure and complete grasp of the identification and expected behaviors of our objects *in the domain where they live*.

Four: hierarchical control and centralization, the epitome of structured programming and traditional software development, must be replaced with as a model of distributed coordination and cooperation.

Consider one of the more ubiquitous models employed in traditional software development, the program structure chart, **Figure 3.1**, popularized by Meillor Page-Jones. At the top of the chart is the puppet-master module, attended-to by a court of special purpose input, transform, and output modules. The puppet-master incorporates all the knowledge about the task at hand, the capabilities of each subordinate module, and when and how to invoke their limited capabilities. The same thinking characterizes structured source code where a main-line routine (frequently a CASE statement) consolidates overall control. A collection of special purpose subroutine paragraphs are individually invoked and given limited authority to perform before control reverts back to the mainline.

F03xx01

Figure 3.1
Program Structure Chart

Unlike puppet modules, objects are autonomous. They are protected from undue interference and must be communicated with, politely, before they will perform their work. It is necessary to find as a different means to coordinate the work of objects, one based on intelligent cooperation among them.

It is sometimes difficult to conceive how coordination among autonomous objects can be achieved without as a "master controller or coordinator." One simple example is the common traffic signal. Traffic signals coordinate the movement of vehicles and people but have no awareness of what those other objects are about or even if any of them actually exist. A traffic signal knows about its own state and about the passage of time and how to alter its state as a function of elapsed time. In this model the necessary "control" has been factored and distributed. The traffic signal controls itself and notifies (by broadcasting as a different color) others of the fact that it has changed state. Other objects, vehicles, notice this event and take whatever action they deem appropriate according to their own needs and self-knowledge.

But, what about intersections with turn arrows that appear only when needed? Who is in control then? No one. Sensors are waiting to detect the “I am here” event from vehicles. The traffic signal is waiting for the sensor to detect that event and send it a message, please add turn arrow state.” It add the state to its collection of states and proceeds as before. The sensor sent the message to the traffic signal only because the traffic signal had previously asked it to – registered to be notified of the “vehicle present” event. Traffic management is a purely emergent phenomenon arising from the independent and autonomous actions of a collectivity of simple objects – no controller needed.

Eliminating centralized control is one of the hardest lessons to be learned by object developers.

Object Principles – Software Principles

Stating and explaining object presuppositions is important. It is also important to show the relationship between those principles and generally accepted principles of software design criteria. Exploring that relationship will further explain and illustrate the object principles and show how they recast thinking about design without rejecting traditional design goals.

Witt, Baker, and Merritt have written an excellent encapsulation of the fundamental ideas about software design and architecture[5]. One chapter identifies a set of generally accepted axioms and principles that define software quality.

- Axiom of Separation of Concerns – solve complex problems by solving a series of intermediate, simpler problems.
- Axiom of Comprehension – accommodate human cognitive limitations.
- Axiom of Translation – correctness is unaffected by movement between equivalent contexts.
- Axiom of Transformation – correctness is unaffected by replacement with equivalent components.
- Principle of Modular Design – elaborates the axiom of separation of concerns.
- Principle of Portable Designs – elaborates the Axiom of Translation.

· Principle of Malleable Designs – provides the means for compositional flexibility.
· Principle of Intellectual Control – appropriate use of abstractions
· Principle of Conceptual Integrity – suggests a limited set of conceptual forms.

Few would argue with these axioms and principles, although they would certainly argue about the appropriate means for realizing them. Object thinkers strive to achieve the goals implied by these axioms and principles as much as any other software developer and believe that objects provide the conceptual vehicle most likely to succeed.

An object is the ultimate form of modular design. It provides a conceptual unit simple enough to satisfy any constraints imposed by human cognitive limitations. Properly conceived an object is a natural unit of composition as well. An object should reflect natural, pre-existing, decomposition ("along natural joints") of a large-scale domain into units already familiar to experts in that domain. Conceived in this fashion, an object clearly satisfies the principle of intellectual control. Objects will also satisfy the principle of conceptual integrity because there will be a limited number of classes of objects from which everything in the domain (the world) will be constructed. In Chapter 4 – Metaphor: the Bridge to the Unfamiliar, an argument will be presented suggesting that the total number of objects required to build anything is less than 2,000.

Objects are designed so that their internal structure and implementation means are hidden – encapsulated – in order to satisfy the axiom of transformation and the principle of portable designs.

The principle of malleable designs is potentially the most important of all the axioms and principles presented by Witt, et. al., because it expresses a central, but somewhat implicit, motivation behind object thinking. Building on the arguments presented in Chapter Two - Philosophical Context, it is possible to assert that object thinkers value designs that yield flexibility, composability, and accurate reflection of the domain above all others. Not machine efficiency, not reusability. In fact, it might be said that, for object thinkers, all the other axioms and principles provide the means for achieving malleability; and, that malleability is the means whereby the highest quality software, reflective of real needs in the problem domain, can be developed and adapted as rapidly as required by changes in the domain.

Fred Brooks' wrote one of the most famous papers in software development, "No Silver Bullet: Essence and Accidents of Software Engineering.[6] In that paper he identified a number of things that made software development difficult and separated these into two categories, "accidental" and "essential."

Accidental difficulties arise from inadequacies in our tools and methods, and are solvable by improvements in those areas. Essential difficulties are intrinsic to the nature of software and are not amenable to any easy solution. The title of his paper refers to the "silver bullet" required to slay a werewolf – making the metaphorical assertion that software is like a werewolf, difficult to deal with. Software, unlike a werewolf, cannot be “killed” (solved) by the equivalent of a silver bullet.

Brooks suggests four essential difficulties:

- Complexity – software is more complex, consisting of more unlike parts connected in myriads of ways, than any other system designed or engineered by human beings.
- Conformity – software must conform to the world rather than the other way around.
- Changeability – a corollary of conformity, when the world changes the software must change as well, and the world changes frequently.
- Invisibility – we have no visualization of software, especially executing programs, that we can use as a guide for out thinking.

He also investigates potential silver bullets (high level languages, time sharing, AI, etc.) and finds all of them wanting. Object-oriented programming is considered as a silver bullet and dismissed as addressing accidental problems only.

Although I would agree with Brooks in saying that *object technology* – languages, methods, class hierarchies, etc. – only address accidental problems, *object thinking* does address essential difficulties, and does so with some promise.

Object thinking suggests we deal with software complexity in a manner analogous to the ways humans already deal with real world complexity – behavior based classification and modularization.

Object thinking is focused on the best means for dealing with conformity and changeability issues – the malleability principle – as a kind of "prime directive." And invisibility is addressed, not with an abstract geometry as suggested by Brooks, but via simulation – direct, albeit metaphorical, simulation of the real world. If we can understand the complex interactions of objects in the real world (and we do so every day) then we should be able to visualize our software as an analogous interaction of objects.

Cooperating Cultures

Arguing for the existence of an object paradigm or object culture is not and should not be taken as an absolute rejection of traditional computer science and software engineering. It would be foolhardy to suggest that nothing of value has resulted from the last fifty years of theory and practice.

Claiming that there are clear criteria for determining if software is object oriented is not the same as saying all software should be object oriented. Different problems do require different solutions. But, if the problem (and almost all application oriented software is such a problem) requires an object solution it should get one – not some kind of hybrid or *faux* object application.

Traditional approaches to software – and the formalist philosophy behind them – are quite possibly the best approach if you are working close-to-the-machine - e.g. device drivers or embedded software. Specific modules in business applications are appropriately designed with more formalism than most. One example, the module that calculates the balance of my bank account. A neural network, on the other hand, might be more hermeneutic and object-like in part because precision and accuracy are not expected of that kind of system.

Traditional methods, however, do not seem to scale. Nor do they seem appropriate for many of the kinds of systems being developed fifty years after the first computer application programs were delivered. Consider the simple graph in **Figure 3.4**.

F03xx04

Figure 3.4
Applicability graph

The horizontal axis represents the spectrum of activities involved in systems modeling and application development. It ranges from Analysis (with the accompanying tasks of comprehending the real world, making useful abstractions, and decomposition) to Implementation (compiling, testing and executing).

The vertical axis opposes the Deterministic world (the domain of hardware, discrete modules, algorithms, and small scale formal systems) to the Natural world (businesses and organizations, societies, composite systems, and cultures).

A diagonal bisects the graph to demarcate two realms. To the lower right is the realm where mainstream computer science and formalist ideas have demonstrated success. Emphasis in this realm is on defining and building hardware and using as a finite set of representations (binary and operation codes) and manipulation rules (the grammar of as a compiler) to implement software designs. This is the realm of formalism where systems may be complicated and even large but they are not complex[7].

At the upper left is an area that is largely *terra incognita* as far as computer scientists are concerned. This is the realm of social, biological and other complex, (as that term is coming to be understood) systems. Meaning, in this realm, is not defined - it is negotiated. Rules are not fixed but are contextual and ephemeral. Constant flux replaces long-term consistency. This is the arena that hermeneutic and object ideas offer an expansion of our ability to model and build systems capable of interacting with the natural systems in which they will have to exist.

This is the realm where objects and the object paradigm should dominate.

Objects provide a foundation for constructing a common vocabulary and a process of negotiated understanding of the complex world. Behavioral objects offer a decomposition technique that will yield adaptable constructs for building highly distributed, “intelligent,” and flexible computer artifacts and computer-based systems.

The “informal” methods of CRC cards and scenarios provide the perfect vehicle for negotiating the understanding and meaning required if we are to integrate artificial computer-based systems with natural organizational and cultural systems. Objects provide as a foundation for as a method that will provide rigor (without misapplied formalism) to the process of system and artifact design. Objects offer as a means to instantiate, extend, and implement hermeneutic ideas.

All of this, once we learn Object Thinking.

[1] West, David. "Enculturating Extreme Programmers," Proceedings XP Universe, Chapel Hill, NC. 2001.

[2] Taylor, David. *Business Engineering with Object Technology*.

[3] Jacobson, Ivar. *Business Re-Engineering with Objects*.

[4] Alexander, Christopher. *Notes on the Synthesis of Form*.

[5] Witt, Bernard I., f. Terry Baker, and Everett W. Merrit. *Software Architecture and Design: Principles, Models, and Methods*. Van Nostrand Reinhold. 1994.

[6] IEEE Computer, April, 1987.

[7] Complex is used here to denote systems that exhibit non-deterministic behavior, that are self-organizing, and which have emergent properties.