4

Metaphor: Bridge to the Unfamiliar

Along the philosophical fringes of science we may find reasons to question basic conceptual structures and to grope for ways to refashion them. Old idioms are bound to fail us here, and only metaphor can begin to limn the new order.

Quine

Explanations without metaphor would be difficult if not impossible, for in order to describe the unknown, we must resort to concepts that we know and understand, and that is the essence of metaphor.

MacCormac

Object philosophy generates a different view of the world, one that is strange to most and especially to developers trained in conventional methods and ideas. This raises the question of how best to assist developers to understand the "new world of objects."

In other disciplines metaphor is frequently used to help those new to an area of study comprehend its fundamental concepts.

Consider the Bohr model of an atom, as one example. Physicists and chemists need to explain atomic structure to lay people and to new students. One common way to do so is to employ Bohr's metaphoric model that says an atom is like a tiny solar system - a nucleus (sun) surrounded by orbiting electrons (planets). This metaphor is technically wrong of course but remains a useful tool for introducing atomic concepts.

Unsurprisingly, metaphors have also been used to convey object concepts. One of the earliest, coined by Brad Cox, is the "software IC (integrated circuit)." This metaphor juxtaposes a desirable trait for software objects with hardware components, that is the ability to use them as standardized and interchangeable parts to “mass produce” larger constructs. It is possible, for example, to shop at a number of electronics stores, buy standard components from a variety of manufacturers and use those components to assemble a working personal computer. Construction of software in a similar manner is one goal of object-orientation, hence the applicability of the metaphor.

Behind the Quotes

Brad Cox

Brad Cox was one of the earliest advocates of object oriented programming and was the developer of the Objective-C programming language that was the core of the Next Computer operating system (Next was the first company started by Steve Jobs when he left Apple Computer) and development environment. Objective-C was Smalltalk with the efficiency and power of the C programming language.

Dr. Cox coined an early metaphor for object components – software IC's – suggesting that software components should be as modular and as composable as the integrated circuits (IC's) used to build hardware. Given an appropriate set of IC's, software could be mass produced – cheaply and with quality – just as guns were, after Colt and Remington invented standardized parts for guns.

One of Dr. Cox's most intriguing ideas was the concept of "Superdistribution." The core idea of superdistribution was to make individual software modules available via the Web and charge for their use, object by object – a concept that is evident in Microsoft's vision of Web services.

Dr. Cox recently resigned from George Mason University to concentrate on a company that will make superdistribution a reality. He has many other interests and his Middle Of Nowhere (*http://www.virtualschool.edu/mon*) is well worth a visit.

The software IC metaphor is less helpful, however, when our concern is object discovery and specification because it only tells us of a desirable trait for the finished product. Alan Kay, Adele Goldberg, Ken Rubin, and Phillipe Kahn, among many others, employ the metaphor of a person when engaged in the process of discovery and analysis. An object is like a person. This metaphor is sufficiently important to object thinking that it warrants its own special term – anthropomorphization. And, of course the more syllables a word has, the more important it must be. We will return to this metaphor later.

Metaphors are not just a tool to explore the unfamiliar. Metaphor is essential to everyday thinking as well. The full importance of metaphor in shaping our thoughts has received a lot of attention in recent years, notably in the work of George Lakoff.

Behind the Quotes

George Lakoff

Professor Lakoff teaches and researches linguistics, cognitive science, and cognitive philosophy at UC Berkeley. His books most likely to be of interest to readers of this book include: *Metaphors We Live By*; *Women, Fire, and Dangerous Things*; and, *Philosophy in the Flesh*. Lakoff's work is frequently cited by those criticizing traditional approaches to software development – especially the kind of set based category theory underlying traditional approaches to data and data modeling – as well as those advocating a more human-centric approach to computing and the software development process.

Dr. Lakoff 's research reveals the central role of metaphor in all human cognition and provides a foundation for Kent Beck's ideas about metaphor in extreme programming – everything from the value of a system metaphor (in lieu of architecture), to the need for metaphorical awareness when creating object and method naming conventions.

Metaphor shapes our thinking in many different ways.

- It helps in discovery – if our system is like an "X" then this component is probably like a "Y".

- It helps us make design decisions – "when a person is asked for id they usually hand over some sort of document (drivers license, etc) so our object probably should store identifying information in some other kind of object instead of in instance variables.
- They provide handy ways to remember principles of object thinking – "objects are naturally lazy and this is starting to look hard, we had better refactor our design and split this work up among several objects."
- They help us avoid old ways of thinking by avoiding the metaphors that are associated with that kind of thinking – instead of "next the machine needs to do this" we use "just ask object X to do that."

Metaphor plays a critical role in XP as well as in object thinking. Kent Beck used his keynote address at OOPSLA 2002 to explore all the ways that metaphor affects all aspects of XP. One of the twelve practices in XP is the system metaphor which is deemed powerful enough to eliminate the need for detailed up-front architectural design to guide development. In Kent's book on Test Driven Development[1] he talks about how different metaphors led to several different designs and implementations of "multi-currency money."

Metaphors are very powerful. Object thinking is absolutely dependent on selecting and employing the "right" metaphors. Each metaphor – whether general or specific to design details – must be consistent with the philosophy behind objects as discussed in previous chapters. In this chapter we will introduce several key metaphors.

The Lego Block Metaphor

Our first metaphor helps us think of important object characteristics, like composability, simple interfaces, and comprehensibility (limited number of forms). It

also illuminates important aspects of the object-oriented software development process – most notably, the fact that there are two distinct though related processes required.

Let's express the metaphor as a dictum: *software should be assembled from a finite set of composable units the way that dinosaurs, and castles, and spaceships are constructed from a common set of Lego Blocks.* It is no accident that the first special issue of the Communications of the ACM devoted to objects had a cover photo of stacked red and yellow Legos.

On its face this seems to be a restatement of Brad Cox's software IC metaphor. But there are depths and nuances to this metaphor that are missing from the simpler 'software IC' idea. Like Cox's metaphor, this one tells us that we should be able to construct an arbitrary number of software artifacts from a finite set of standard parts. This is a characteristic of many other aspects of nature as well as mass produced products. For example: the world around us constructed from a very finite set of parts, atoms of the periodic table or below them the quarks, especially given the variety and complexity of the things manifest in the World. Houses are constructed from 2x4s, nails, 4x8 sheets of plywood, etc. Even societies are built from, "butchers, bakers, and candlestick makers."

By focusing on the composable nature of Lego Blocks, the metaphor reminds us of how object thinking views the importance of decomposition and composition (remember Plato's views quoted earlier). The metaphor also reminds us of the importance of reuse. Reuse in the sense that the same block can be used in many different contexts. Reuse in the sense that you can remove a block from a dinosaur and recycle it in the construction of a space station.

More importantly perhaps, it reminds us of the fact that simple and obvious interfaces are required if our objects are to be as composable and as useful as Lego Blocks. Further exploration of the metaphor and its implications suggest ways to discover, design, and build truly reusable and composable objects. This alone would make the metaphor extremely valuable. Given that developers have pursued the dream of reusable code libraries from the very advent of computing – with very limited success – a way to actually accomplish that goal would be invaluable.

Exploration of the metaphor begins with considering what it implies about development process – the suggestion that there is a necessary separation between the process required to create objects (Lego Blocks) and the process of assembling those objects into useful products (software, in our case). The metaphor suggests that:

- Creators are (probably) adults working for the *Lego Corporation*.
- Users are children (at least at heart), ages four to adult.
- Creators have specific concerns and use specialized processes to accomplish their goals
- Users care little about the components, as components. Their concerns focus on what can be built with the components and the ability of the artifact to satisfy their needs.
- The component 'engineers' need to be very concerned with the internal structure of the blocks, what kind of plastics will yield the correct degree of malleability, colorfastness, friction to keep them together, spacing of the pips at the top of the block, etc. etc. They also have to create components that transcend particular applications because their goal is to build blocks equally useful for dinosaur and spaceship construction. More importantly they want blocks that can be used successfully by an unknown end user to build whatever it is that they have imagined.

- Users want to easily move from concept to construction without the need to concern themselves with technical details. They want rapid feedback, they want to be able to change their mind mid-construction, and they want the artifact constructed to operate in their world as it is.

Applied to software the metaphor suggests separation of domain decomposition and object definition from the tasks of assembling applications and solving specific operational problems. To some degree, with some implementation languages, this separation has started to occur. Consider the class libraries that come with a language like Smalltalk or Visual Basic. Many of the classes (the collection and magnitude classes, for instance) in such libraries reflect the same kind of general and abstract thinking that leads to good "software *Legos*." Another example is the attempt to create visual programming environments for application assembly that are, at least quasi-independent of the underlying implementation language. Such attempts are but a start towards an "object mature" world where the two tasks are as clearly separated as they are in the world of *Legos*.

I would like to bring to the reader's attention a nuance of this metaphor as "something to think about" without attempting to fully develop the idea. The metaphor suggests differentiation between users and creators that, in the case of the Lego Block is, is very different from the similar distinction made in this book. A Lego user is a child – a kind of ultimate consumer. In the case of objects and software, we are treating other programmers as users.

If we were to be completely consistent with the Lego metaphor we would have to argue in favor of delivering objects to end-users, those filling roles in business and organizational worlds, and not to programmers. The "objects" would have to be directly usable without the need to use programming environments and compilers.

> Each component would be need to be a small executable program, modifiable via user messages; not modules of source code made available to programmers.

Another, critically important, aspect of the metaphor is the ability of users to successfully employ the blocks based solely on their intuitively obvious external characteristics. A child can look at a block and instantly tell if it is suitable for inclusion in the project at hand. It is not necessary to know anything about the chemistry of plastics or whether this particular block was made at "Sun Lego Corporation" or "Microsoft Lego Corporation." There is no need to read a complex user manual that explains either the block or how to use it.[2]

Software objects cannot even approximate this degree of composability, but the metaphor suggests that full realization of object potential requires satisfaction of this characteristic.

The fact that two different groups of people, and two different processes, are involved in block creation and artifact assembly, one group adults and the other children might lead one to believe that the metaphor de-skills the task of application assembly - after all it can be done by children (or end-users, perhaps). This would be a misleading conclusion.

The *LegoLand* store in the Mall of America (where I first encountered one) periodically sponsors two events. In the first, children are invited to use the unlimited set of blocks at hand to construct various things. Prizes are given to the best constructions.

The second invites professional architects and designers to use the same blocks to create various structures that were then sold as part of a charity auction.

As should be expected there was a large qualitative difference between the constructions of the architects and those of the children. The building blocks remained the same. The architects, however, were able to bring to bear other skills, proportion, geometry, aesthetics, etc. that the children did not yet possess. The architects were domain experts (end users) that were able to use the blocks to build solutions that fully exploited their domain knowledge. They were able to use the blocks to simulate the way they wielded girders and bricks in the real world domain where they worked.

Here too software objects do not come close to providing the "language" sought by Kristen Nygaard or the interactive milieu sought by Alan Kay.

> Following up the immediately preceding note – perhaps the users of objects would not have to be the ultimate end-user, but could be a new kind of professional "assembler" or "collage artist" with a set of skills not available to the end user, but quite different from those required by traditional programmers and software developers. The demand that objects be run-time modifiable executable programs would not change.

Two other items suggested by the metaphor are related. Objects should be simple and there should be relatively few of them. There are less than 10 basic *Lego Block* types. (Kits contain additional parts, each of which is highly specialized, like tiny human figures and motors, but these cannot be considered true Legos.) There are only 134 elements and only six quanta. The vast majority of houses in this country are built with

less than ten standard sizes of dimension lumber. In all of those cases as well, the base elements are simple and highly specialized.

How Many Objects?

One of the heuristics for object discovery is, "find the nouns." Each noun in a domain description is a potential object class. Estimating the "minimal class set" could involve a simple count of nouns employed in a domain.

Expanding on this heuristic – how many classes would be required to model the Universe? Well, how many nouns are required?

The Oxford English Dictionary has about 550,000 words. An English teacher once told me that about 40-45% of the words in a dictionary will be nouns.

If we eliminate proper nouns and synonyms that percentage will be reduced to around 25-30% of the words describing potential objects. Eliminating archaic nouns – bodkin and amanuensis, for example – will reduce the percentage still further, 15-20% perhaps. This translates into about 110,000 classes. A pretty large number but clearly finite.

Instead of the OED, however, a better estimate might be obtained using the vocabulary required to read the average daily newspaper. Most things are quite adequately described in a newspaper.

Vocabulary required to read a typical newspaper – about 1,400 words!

Using the 30% estimate (we don't have archaic terms to eliminate, hopefully) this suggests a need for only 420 classes. Allowing liberal ability to add classes

representing objects that do not make the paper and we still come up with less than a thousand classes to model all typical domains of human interest.

If your problem domain – or worse, your application – has thousands of classes (and I have seen some) then you probably have yet to master object thinking.

This suggests that there is a similarly small number of objects from which we can construct any type of software needed to model any domain or organization. (See sidebar, How Many Objects?)

It is frequently convenient to build a large construct from smaller, but not the smallest possible components. It is easier to build living things with hydrocarbon molecules than directly with individual atoms. It is easier to build a roof with a truss made of 2x6 boards and gang nails than a board and a nail at a time. Intermediate constructs, like trusses used to build houses, are components. The number of components will be much larger than the number of objects from which those components are constructed. Moreover, they will likely reflect stylistic differences reflective of the designers and potential users of such components.

The last item suggested by the metaphor deals with process. Watching a child work with the blocks reveals a process of discovery filled with a certain amount of trial and error and supportive of rapid change as prototypes fail to meet satisfaction criteria and so are taken apart and reassembled in another attempt to reach the envisaged goal.

XP development closely resembles "playing with Lego Blocks" in the sense that it too allows discovery and emergent solutions, tolerates and leverages mistakes, encourages taking things apart and reassembling them into more elegant solutions (refactoring), and relies heavily on feedback as to the extent to which the current assembly meets user expectations.

Both XP and the metaphor suggest a need for a development environment that supports this kind of development process model. Smalltalk and visual programming environments, like *Visual Studio*, provide examples of development environments and tools that are superior to, in this regard, "compile-link-test" environments like C++ (even Visual C++ with incremental compilation). This notion would seem to be born out in experience. It typically takes about half the time to develop an application with Smalltalk than C++, given equivalent levels of skill in the developers, even with current incremental compilers. Even advocates of languages like C++ will tend to cede the speed-of-development issue and focus instead on characteristics-of-product issues like speed of execution.

Object as Person Metaphor

People are objects and objects should be conceptualized as if they were people. Projecting human characteristics onto inanimate things is called anthropomorphization. Anthropomorphizing in a philosophy class might earn you a poor grade but it is essential to a good understanding of objects.

Phillipe Kahn starred in, *The World of Objects*, a video he produced to illustrate object concepts. He used musicians, a specialized type of human being, to illustrate the anthropomorphization of objects. Mr. Kahn noted that he is capable of playing numerous instruments and splicing the results together to create a finished piece of music. Better results with less work, however, would be obtained by engaging a group of talented musicians - each of whom was a specialist in some instrument or aspect of music. He could then communicate his desires to them, allowing them to respond to those desires using their innate skills, knowledge and abilities. The musicians would be asked, at a relatively high level (asking the percussionist for some "color," for example) for some service. They in turn would interpret (make sense of) those instructions in terms of their own abilities, experience, and even awareness of the context in which the request was made, then respond in an appropriate manner.

Behind the Quotes

Phillipe Kahn

Phillipe Khan was the founder of Borland International, once one of the largest software companies in the world and leader in the field of integrated development environments (IDE) that combined programming language, editor, incremental compiler, debugger, and visual tools to simplify and speed up software development.

Borland was an aggressive early adopter of object ideas for its own software development and enjoyed numerous early successes with object technology. Mr. Kahn left Borland, and the world of the PC, to concentrate on network applications as CEO of Starfish Software. His current work still reflects a commitment to object ideas.

This metaphor tells us that an object needs to be an agent capable of providing a specified set of services. It has access to a body of knowledge (some of it internalized) that it uses to respond to our service requests as well as any necessary mechanisms (talents and skills) and resources (instruments, a computer, or whatever else it may need). It also tells us that the (only) appropriate way to determine if an object will suit our needs is by a careful review of its resume.

Applying this metaphor can feel a bit strange at first. What, for example, are the talents or skills possessed by a book? What services can, or does, it provide? What knowledge does it require in order to respond to requests for those services? What resources does it need? These appear to be hard, or nonsensical, questions to answer. But with some practice, a book is revealed as:

- A collection of page objects.
    1. Working with those page objects it maintains order (sequence).
- An object capable of identifying itself.
- An object that can describe itself.
    2. The description being an separate object which can provide individual elements of the description – like date published, author(s), publisher, etc.
- An object than can provide the reader with access to a specific page or a group of pages (chapters) upon request. (Also in collaboration with the page or chapter objects.)
- An object that acts as a "front" for a community of objects (pages, tables of contents, indexes, chapters, etc.) allowing users of the community a convenient point of

contact. Requests can be sent to the book, knowing that the book will relay those requests to the actual objects capable of responding, without interference.

- At one point in its life (in the days it was a mere unpublished manuscript), it also was able to add and delete pages to its collection and still maintain the proper order.

If objects are persons, they are limited in some of the same ways as human persons. One example, they need to know certain things in order to complete an assigned task. When the book was young and still being composed (playing the role of manuscript), it might be asked to add a page to its collection of pages. In order to do this it needs to know the page to be added.

Like people, software objects, are specialists. They are also lazy. A consequence of both these facts is the distribution of work across a group of objects. Take the job of adding a sentence to a page in a book. While it might be quite proper to ask the book, "please replace the sentence on page 58 with the following ' …'." (The book object is kind of a spokesperson for all the objects comprising the book.) It would be quite improper to expect that the book itself actually did the work assigned. If the book were to do that kind of work it would have to know everything relevant about each page and page type that it might contain and how making a simple change might alter the appearance and the abilities of the page object. Plus the page might be offended if the book attempts to meddle with its internals.

The task is too hard (lazy object) and not the book's job (specialist object) so it delegates – merely passes to the page object named #58 the requested change. It is the page object's responsibility to carry out the task. And it too might delegate any part of it that is hard – to a string object perhaps.

Following these patterns we let the book and the page do what each does best and what is appropriate for each. If we need a service that requires a contribution from more than one object we either assume responsibility for asking the objects for their individual contributions and assembling those results in a way that suits our purpose; or, we send the request to whichever object is acting as spokesperson for the group and allow it to delegate tasks and assemble responses in order to reply to our request.

The person metaphor guides our decomposition and our assignment of responsibilities to software objects that, always, reflect the demands of the domain, not their eventual implementation. Our software object should simulate the services provided by our real world object, both of which we metaphorically regard as "people." Even though it is true that we must be more precise in specifying our software object, it is critically important that we continue to use the real world and not the computer (implementation) world as the foundation for our conceptualization of the software object.

We said that objects have access to all of the resources necessary to do their jobs. In the case of a software object this means that each object is assumed to have access to all of the resources of an arbitrarily complex computer system if necessary. This is one reason that Alan Kay calls objects, "intelligent virtual computers." Because we are conceptualizing each of our objects as possessing its own computer (virtual computer or thread) we have the foundation for concurrent or parallel processing systems. Objects, like the people we metaphorically equate them too, can work independently and concurrently on large-scale tasks, requiring only general coordination. When we ask an object collective to perform a task, it is important that we avoid "micro-management" by

imposing explicit control structures on those objects. You don't like to work for a boss that does not trust you and allow you to do your job, why should your software objects put up with similar abuse?

Variations on the object-as-person metaphor include the object-as-agent metaphor. This is really the same idea but focused on seeing the object only in terms of the services it provides on a particular client's behalf. The only difficulty with this variant metaphor is the possibility of creating a lot of agents defined by client needs instead of finding a general agent abstraction that can serve the needs of multiple clients. The latter will be a more composable object and is far more reflective of how we view people – you can do your job on behalf of many different employers in widely divergent businesses. Design your "object people" with the same capability.

Earlier we compared a kindergarten teacher, as classroom administrator, to a book. We assigned a "role" to the teacher. In the real world we think nothing of people fulfilling multiple roles. People are actors and objects can also be actors. This metaphor leads into an entire category of theatrical extensions, (some discussed below). It means that a single object can appear quite differently in different contexts. Mel Gibson can be Hamlet or The Road Warrior, two distinctly different roles. A collection object can be used to replicate the behaviors of a book or a parking lot, depending on the context and the values and objects manipulated as it performs its services.

Software as Theater, Programmers as Directors

Engineering conveys an image of bridge or building construction. Software engineering is therefore a metaphor for how software should be constructed, i.e., in a manner analogous to constructing a physical structure. Formalists love this metaphor but object advocates find it counterproductive. A better metaphor for assembling objects to collectively perform tasks is, "theater."

Software development is analogous to casting and directing a play. First task is to select a your players – the objects that will collectively complete the expected tasks.[3] Provide the cast with a script (cues and dialog). Test them (practice or rehearsals) to make sure you have the right actors and the right cues and dialog, then, when satisfied, put them on stage and raise the curtain. If you have done you job well the actors will proceed through the play and the audience will be provided the service of entertainment.

Most software development involves the recreation of an "old standard" an original play that was already cast and performed in the real world using human and tangible objects (actors). Software developers face the same challenge as stage directors – how to make the play innovative and fresh without alienating the audience by removing too much of the familiar and expected.

As fanciful as this metaphor may appear, it is deadly serious. And it reveals one of the flaws of traditional software development – focus on the artifact instead of the system in which the artifact will operate. Software (computer) artifacts that fail to simulate, in a reasonable familiar manner, and that are unable to interact with the other objects in the real world that the software artifact will operate will fail. Software development is reality construction (or reality reconstruction) just as Floyd and her

colleagues have asserted. And the theater metaphor helps us accomplish our task by reminding us "that all the world is a stage" and that our artifacts are but actors on that stage.

Occasionally developers have a chance to create a brand new reality. Most of what we get paid for, however, is simply reproducing standard works. Sometimes that involves replacing a single actor – like casting Mel Gibson as Hamlet instead of Laurence Olivier. Some times it involves an almost complete surface change – like the Star Trek episode that basically restages *Moby Dick* in outer space. (Picard as Ahab, the Borg as the white whale.) The theatre metaphor adds the concept of verisimilitude (the appearance of being real) as a criterion for good software and system design.

Sometimes the script followed by our object actors will be fixed. In software, most batch processes would be considered to have a fixed script. But more and more software needs a mix of fixed and extemporaneous scripts. Extemporaneous scripts are those that are highly interactive and where the conversation is not predictable in advance.

Visual programming environments provide an illustration of this metaphor. Object (cast) selection is accomplished by dragging iconic representations from a catalog to a workspace. Links, between or among, objects are established by drawing lines to connect them with each other. Each link defines a circumstance in which one object communicates (events and messages are two types of communication) with another. The collection of links established in the workspace is the script for that group of objects.

The play (theatrical production) metaphor can be extended a bit further. Plays come in many sizes and the complexity of the script varies accordingly. A one person play being relatively simple to produce (but one that requires an exceptionally talented actor) while a Cecil B. DeMille epic with a "cast of thousands" is considerably harder to

organize. The complexity of object oriented application software is in the scripting, not the objects.

Our tools, as applications software developers, for dealing with this complexity are still quite limited. One need only reflect on the rapid accumulation of visual clutter, (overlapping lines, obscure icons) in a visual programming work space to see how limited our ability is to describe large-scale interactive scripts. Like any other complicated task, we will attempt to solve this problem by decomposing our play script into act and scene scripts.

Three additional aspects of this metaphor deserve some discussion before moving on. First, although we classify plays as being of various types depending on their complexity and scope, all theater is essentially the same, a group of actors focused on accomplishing a particular objective, coordinated by a script. With software we distinguish between objects, components, applications, subsystems, and systems for our convenience. Close examination reveals that in each case we have a number of objects focused on accomplishing a small list of tasks while constrained by a guiding script. This tells us that we need to apply the same principles of object philosophy whether we are constructing the most specific class or a system with wide scope. We do not suddenly revert to old habits just because the job is larger.

Second, replacing actors of similar talent and skill set should not require a rewrite of the script for a play. Substituting objects capable of the same behavior should not require a redesign of the software. We should be able, in fact, to nuance or dramatically change the overall behavior of our software simply by changing players. A drama can be

turned to a comedy simply by replacing dramatic actors with comedic actors. The latter receive the same cues and deliver the same basic behavior (say the same lines) but use their innate abilities to interpret the cues and respond in very different ways. This point will become important later when we discuss application frameworks.

Third, just as plays are categorized by genre it is appropriate to extend the theater metaphor to software and classify systems into genres, based on typical forms of organization or architectures. An architecture as a patterned way of organizing a set of actors, as is a genre. We generate expectations and constrains that will apply to our actors based on the genre of the performance, or the type of architecture. Architectures also provide general solutions or frameworks that make it easier to conceptualize the organization of our cast. Patterns (genres) provide 'script templates' to which the designer adds detail in order to construct the actual script used by an object collective to complete its work.

> Architectural patterns will be discussed in Chapter Nine, Objects on Stage. "Pipes and Filters," "Model-View-Controller (MVC)," "Blackboards," and "Client Server" are but a few examples of software architectural patterns that will be discussed.

Care must be taken, however, to make sure that assumptions implicit to the architecture (genre) do not contradict or contravene object thinking. It would be difficult, for example, to use the typical "Hierarchical Control" script that is embodied in the infamous program structure chart popularized by Yourdon and Page-Jones in an object fashion.

Ants, Not Autocrats

Control is anathema in the object paradigm. It is replaces with a kind of "blind coordination" as exemplified in the traffic signal example in Chapter 3, From Philosophy to Culture. A traffic signal is "blind" in the sense that it does not need any awareness of other objects or their goals to accomplish its own tasks. Any sense of traffic control has been distributed to the collection of objects in the intersection and not to any single object.

The traffic signal assumes responsibility for monitoring the passage of time and cycling through a change of states at appropriate intervals. (Green for 20 seconds, Yellow for 10 seconds, Red for thirty minutes[4].) It also assumes responsibility for notifying others of its current state by broadcasting that state via an externally observable colored light. Automobile (or driver) objects assume responsibility for inhibiting or expressing their own behavior (Stop on Red, Go on green, Accelerate on Yellow[5]) as a consequence of their awareness of the signal. Neither automobiles nor drivers know anything about the workings of traffic signals just as signals know nothing about automobiles or drivers.

This kind of blind coordination seems to work well in small-scale examples like the traffic signal, but does it "scale up?" The answer, suggested by ants, termites, and biological communities, is yes. "Hive communities" collectively construct extremely elaborate structures, efficiently exploit natural resources (like food) without the need for architects or overseers. No single ant is in charge of making sure that a group of ants perform. Food foraging begins when a single discoverer ant broadcasts his (the only

female ant, the queen, is back in the hive) discovery to the other ants by exuding a particular pheromone. Other ants detect the pheromone and respond by moving to the food source then back to the hive, also exuding the same pheromone. No ant is aware of the identity of any other ant. They do not seem to care if there are other ants around. They simple detect an event (receive a message) and respond according to their intrinsic nature.

If the idea of patterning software architectures on ant or termite colonies makes you uneasy, you might consider the work of Marvin Minsky in computer based artificial intelligence. Minsky (who many also credit with the idea of OO programming) posits the construction of intelligence by utilizing non-intelligent actor/agents (objects) that make simple decisions based on their own local awareness of themselves and their individual circumstances. These simple agents are aggregated in various ways until the form a "Society of Mind," which is also the title of the book where these ideas are formulated.

> Minsky, however, did not approve of emergent behavior – did not believe it applied to software and to artificial intelligence research. His society of mind relied upon controllers – and to the extent that it did so, implicitly presumed, in those controllers, the very intelligence that he was trying to model with the aggregate society. Software developers attempting to follow object thinking principles find it extremely difficult to avoid the notion of control and sneak in implicit control in lots of subtle ways, just as Minsky's critics suggested he did with his society of mind.

The attempt to establish centralized economies and management in the pre-collapse Soviet Union is a contrasting example of autocratic top down control. It suggests that although it might be possible to construct very small-scale control oriented

systems it does not work on a large scale. The emerging discipline of complexity theory also provides insights into the limitations of control in large systems.

These two counter-examples provide the basis for the metaphor that objects are coordinated as if they were ants and no object attempts to assume the role of autocrat controlling the behavior of other objects.

Inheritance and Responsibility

Inheritance - humans naturally aggregate similar things into sets (or classes). Another 'natural' kind of thinking is to create taxonomies – hierarchical relationships among the sets. The most common example of this kind of thinking is Linneaus' taxonomy of flora and fauna and the subsets of that taxonomy that are general common knowledge. Fido is a dog (example of aggregation and the subsequent identification of a set). The set of Dog is a subset of 'Canine,' which is a subset of 'Mammal,' which is a subset of 'Animal'. (The example taxonomy is neither complete nor is it intended to be accurate; merely illustrative.) We could also say that a dog is-a-kind-of canine which is-a-kind-of mammal which is a-kind-of an animal. Taxonomies are tree structures.

Another kind of tree structure – one that actually employs the term – is a genealogical chart, a "family tree." Because both the taxonomy and the genealogy chart use the same structure, a hierarchical tree, they have become a kind of conflated metaphor.

The terms “parent” and “child,” for example, are clearly appropriate for genealogy but are somewhat suspect when applied in the context of a Linnean hierarchy. Nevertheless, it is common to speak of the super / sub set relationship in terms of “parents” and “children.” A super class is “Parent” and a subclass is “Child.” From here it is but a short step to talk about Children “inheriting” from Parents.

At this point the metaphor can be helpful or potentially misleading depending on how it is used.

According to the presuppositions of the object paradigm a child class is a behavioral extension of the parent. A dog has all the behavior of a mammal plus some additional behavior that is specific to dogs. If we say that a dog inherits the behaviors of a mammal and mean by that statement that a dog be asked to do anything that a mammal can do we are using the metaphor properly.

Too often, however, the metaphor is used to assert that the child class inherits the internals of the parent class - an allusion to the fact that biological organisms inherit the DNA structures of their parents. This is a poor and potentially misleading use of the metaphor.

Behavior is the abstraction that we use to differentiate among objects and should be the only criteria that we use to establish our taxonomy. Using any other criteria will make our taxonomy more complicated at minimum and erroneous at worse. In the real world, errors like racism and sexism can be seen as characteristic-based rather than behavior-based taxonomies: with the obvious negative consequences.

In the context of software objects, creating taxonomies based on internal structure (e.g. attributes or operations) causes problems in numerous ways. One example is the need to create new classes of objects like "Customer" just because it has different characteristics in some contexts than in others (a credit rating, perhaps). It can also lead to an apparent need for multiple lines of "inheritance" when an object has characteristics that are part of the structure of two or more potential parent objects.

The desire for a child class to inherit internals of its parent classes can be better accommodated if we change the notion of inheritance from DNA to assets. It has been noted that an object has access to whatever resources it needs to fulfill its behavioral expectations. If we say that child classes have access to the resources of their parents via inheritance, i.e., by virtue of the parent-child relationship, then our use of inheritance remains consistent with the object paradigm.

Responsibility - we know an object by what it does, what services it can provide. That is to say we know objects by their behaviors. We are not interested, of course, in just any old behavior. We have specific expectations of our objects and brook little deviation from those expectations. Because we expect objects to exhibit certain specific behaviors we tend to hold them accountable for those behaviors. We obligate them to perform as expected. We make them 'responsible' for providing those behaviors, on demand, as services.

An enumeration of an object's behavior can therefore be considered a listing of the object's "responsibilities." Using the metaphor responsibility implies several things

about the object: consistency or contractual obligation, and self-control are two important examples.

If an object states that it is capable of providing a given service it should perform that service in all circumstances and the results should be consistent. If an integer object says, “I can add myself to any integer you provide and return to you the resulting integer,” it would be very irresponsible if sometimes it returned a floating-point number, or worse an integer that was other than the one representing the summing process.

Responsibility implies that an object must assume control of itself. It must be capable of assuming responsibility for its own maintenance, for notifying others of internal changes that they might need to be aware of, making sure it is persistent if it needs to be, protecting its own integrity, and coordinating its use by potentially multiple clients. Just as it is improper for one object to assume a role as controller or manipulator of another, it is improper for an object to fail to assume responsibilities that makes the need for external control unnecessary.

The notion of self-responsibility will play an important role in how we decide to allocate behaviors across a collection of objects.

Thinking Like an Object

Metaphors like those just introduced should shape our thinking about objects and about software constructed from those objects. Internalizing these metaphors and the philosophical presuppositions stated in earlier chapters allow you to start "thinking like

an object." Until you are able to do so and, more importantly, until you can extend them into new areas, you will fail to realize the full potential of objects. In large part the internalization process will occur over time and with experience. It is possible to adopt certain techniques and utilize certain models to enforce the practice while you are gaining experience. Later portions of this book will explore and discuss some of those models.

Most of the models discussed will bear a striking surface similarity to models constructed using classical software modeling techniques. Syntactically the variation between many object and classical models is minimal. It is the semantic content - deriving from how we accomplish our analysis and decomposition of our domain, of the models that will vary.

Because the models are so similar in syntactic structure it will be very easy to revert to old habits, to use the new models in old ways. A partial defense against this tendency is the adoption of a new vocabulary. In the next chapter we will present a vocabulary that reflects both the philosophy introduced earlier and which is consistent with the metaphors presented in this chapter. Whenever the vocabulary definitions seen strange or unduly restrictive – remember the philosophy and metaphor that inspired them.

[1] Beck, Kent. Test Driven Development By Example. Boston: Addison-Wesley. 2003. ISBN 0-321-14653-0.

[2] There are, however, patterns; diagrams suggesting proven ways that you can construct a family of similar artifacts, houses for example. This topic will be looked at in more detail in a later chapter.

[4] OK, only subjectively.

[5] Observed, but highly improper behavior.