5

Vocabulary - Words to Think With

Language shapes our ability to think. Words provide us with representations of particular thoughts and offer a mechanism for communicating thoughts among ourselves. The richness, variability, and subtlety of our thoughts is reflected in the size of our vocabulary. When we discover new things we invent (or borrow) new words to express our discovery. When we want to make critical distinctions between or among similar notions we use specialized vocabularies. The use of special vocabularies is common in our professional lives because, as specialists, we need to become familiar with unique ideas and to make important distinctions that, as laypersons, we may not.

Object-oriented development has its own specialized vocabulary for all of the above reasons. The unique vocabulary of objects has another very important purpose - to differentiate object concepts from traditional software concepts that, *on the surface*, appear to be similar. Object vocabulary is deliberately different and is focused on communicating object thinking. To ignore the vocabulary or to trivialize the differences between object and traditional terms is to make a significant error.

Perhaps the most asked question in an introductory object class is some variation of, "what is the difference between a (class, method, message) and a (module, function, procedure call)?

In one sense this question can only be answered with, "nothing, there is no difference." Given this answer the next question becomes, "why then do you use these silly new words?" Underlying this question, of course, is the sneaking suspicion that the object advocate is a snake oil salesman trying to confuse the innocent customer with a fancy polysyllabic vocabulary.

The "object difference" is not readily apparent at the level of implementation. As noted earlier, once software is in the machine it is, "all ones and zeros." The difference is how we think about the problems and solutions. The difference is most manifest in how we do decomposition and how we decide to distribute responsibilities across a community of objects. A message and a function call may be identical, in syntax and in how they are implemented inside a machine. But who sends the message to whom, how that message is interpreted, and what kind of code will be necessary to respond will be quite different.

Object vocabulary is also different because of the underlying philosophy that motivates thinking in new terms and because we are employing different metaphors for what we are about and what we are building. Different philosophy + new metaphors = object vocabulary.

Object vocabulary is first and foremost a technique to help developers avoid the mistake of thinking about solutions using old mental habits.

Although there are numerous terms employed when discussing objects, not all of them have the same degree of importance. There are relatively few terms that must be understood as fundamental and which embody the majority of object philosophy.

Essential Terms	Supporting Terms	Implementation Terms	Auxiliary Terms
Object	Collaboration and Collaborator	Method	Domain
Responsibility	Class	Variable	Business Requirement
Message	Class Hierarchy	Dynamic Binding	Business Process Reengineering (BPR)
Protocol	Abstract/Concrete		Application
	Inheritance		
	Delegation’		
	Polymorphism		
	Encapsulation		
	Component		
	Framework		
	Pattern		

Table 5.1 – Vocabulary

Other terms are supportive and allow us to say other interesting things about objects or the way software objects might be implemented. Still another category; terms that apply almost exclusively to software objects and these terms can vary or be extended as a function of the implementation language

selected for development. Figure 4.1 lists the vocabulary terms in four categories: essential, helpful, implementation, and auxiliary.

Essential Terms

The following terms have the greatest philosophic import – they embody most of the philosophy that makes the object difference.

Object - the fundamental unit of understanding. We define the world (and the world of software) in terms of objects. Everything is an object! An object is anything capable of providing a limited set of useful services. Metaphorically an object is like a human agent or actor. We decompose the complex world around us in terms of objects and we assemble (compose) objects in various ways so that they can perform useful tasks on our behalf.

Our understanding of an object is, and should be, based on that object's public appearance - it's "family" name (class) and an "advertised" list of services (protocol) it is willing to provide. Every object has a "personal name" which is a unique identifier and a "family name" which identifies the set (class) of similar objects to which the particular object belongs. Most often we speak of the family (or class) name of the object. That name is descriptive of the object and should convey a general sense of what kind of services it might be able to provide. The advertised list of services usually takes the form of a list of syntactic phrases that we can use to invoke the object's services plus a list of potential states of the object. The latter are important because some objects need to know about changes in the state of other objects. Together, these lists constitute our behavioral promise to the world at large. We

(speaking as an object) do not share with the world any notion of our internal structure and certainly no sense of how we do what we do.

Every object has access to whatever knowledge is requires to perform its advertised services. This does not mean that the object 'contains' that information. Information can also be accessed by asking another object, supplied as part of the request, or calculated upon demand.

Traditionally, software was conceived in terms of passive data and active procedures. One of the more famous definitions of a program is, "algorithms plus data structures." This is also called the "pigeon hole" approach to software - passive bits of data sitting in boxes (like those seen in a post office sorting room, which are called pigeon holes) where procedures come to remove a bit of data, transform or use it, and then put it back for the next procedure to access.

Because everything is an object there is no data. One of the principle ideas of traditional software approaches has disappeared. Everything, including characters and integers, is an object responsible for providing specific services. The elimination of passive data has important consequences for object design and for object development methods, topics that will be discussed later in the text.

Therefore, two popular graphic models, **Figure 5.1,** of an object are misleading and should not be relied upon. The "donut" model perpetuates the classic ideas of passive data and active procedures and better describes a COBOL program than an object. The "animated data entity" model also perpetuates the outdated separation of data and procedures and compounds that error by focusing on the distribution of data items - attributes - across a group of objects. Procedures are appended to the objects

containing the distributed attributes. This “data driven” approach to object modeling is quite popular but is not consistent with object thinking.

F05xx01

Figure 5.1 – Graphical Object Models
Donut model (left) and Animated Data Entity model (right)

Another, and essential, way that objects differ from the modules at the heart of traditional software decomposition is that an object represents something naturally occurring in the problem domain, while a module represents a logical aggregation of elements appearing in the solution space. This is another reason that the above graphical depictions are misleading and should be avoided. Both diagrams depict “what is to be built” instead of “what is to be modeled.” Object discovery and specification must be domain driven!

Responsibility - a service that an object has agreed (or been assigned) to perform. Objects are charged with performing specific tasks. Each task is a responsibility. The term responsibility is used to aid us in discovering who (which object) should be charged with task without needing to think about the object's structure. (We should not know its structure.) Responsibilities are characterized from the point of view of a potential client of a service. I ask, "who would I reasonably ask for this service" and my answer determines which object becomes responsible for satisfying that type of service request.

Responsibilities are not assigned on the basis of "attributes" or on the basis of knowledge (data) assumed to be in the object's possession. This is the single biggest difference in definition between "data driven" and behavior or responsibility driven approaches to objects.

Responsibilities are not functions although there is a superficial resemblance. Perhaps the easiest way to differentiate between a responsibility and a function is to remember that the former reflects expectations in the domain – the problem space – while the latter reflects an implementation detail in the solution space – the computer program.

There are four basic types of services that an object might perform.

Maintain and supply on request one or more units of information.

The information, (everything being an object), is an object like a string or a character or a number, the value of which conveys meaning to an observer or user. A person object might agree to provide its identification, which means it agrees to provide you with a string, the value of which you would recognize as a unique identifier.

A more complex example would be an object, a person perhaps, that agrees to provide you with its description. In response to your request this time the object gives you a "description object." A description object can be thought of as a collection of labels and associated values. [Figure 5.2] called a ValueHolder. You would then ask the description object for a particular value associated with a specific label.

F05xx02

Figure 5.2 - A Value Holder

A simple two dimensional array with the contents of the first column restricted to labels and contents of the second restricted to "data object" i.e. strings, numbers, dates, characters, etc.

Perform a computational task.

This is a straightforward responsibility as long as you remember, "computations must be performed by an object on itself." A number object, for example, can add itself to another number[1] but it would be inappropriate for a calculator object to add two numbers together. A calculator is a kind of "control object" and should be avoided, both because it violates the object paradigm and because it would have to be a very large and complicated object, another trait to be avoided.

This notion can be generalized so that a mathematical expression - which is a specific kind of computation - should be responsible for evaluating or solving itself. An abstraction of this service is a class of objects called SelfEvaluatingRule. This class will be discussed in detail in a later chapter.

Report on, or update, its state.

Sometimes an object will change its internal nature, its state, and on those occasions it might be expected to report that change to others. The object should not, however, know anything about potential clients and should only indirectly keep a list of those that might want to be notified of state changes.

Every object should have the ability to accept "registrations" for event notification. This might be done via a Dispatcher object, [Figure 5.3]. A registration message would be sent by an object interested in one of the states you might find yourself in and of which you have indicated a willingness to notify others. That potential client will send you a registration Request [Figure 5.4] consisting of its own name and the message it wants sent as the event notification message.

All requests for registrations are place in the notification queue associated with the specified event. When an object changes state (generates an event) everyone registered for that event is notified.

F05xx03

Figure 5.3 – An Event Dispatcher
Events to be dispatched are located in the first column, the second column is a queue of event registration requests (Figure 5.4).

F05xx04

Figure 5.4 – A Registration Request
A simple dyad – who is to be notified (an object) and the means of notification (a message).

Coordinate other objects.

The key here is to maintain the "blind coordination" principle. Examples might be dispatchers that route messages without any knowledge of why the messages are being sent or queues that coordinate the sending of a group of messages and hence the activation of the objects receiving those messages but does so only because its nature is to maintain order among a collection of objects and release them from the queue at appropriate times.

The same dispatching objects (Dispatcher and Registration) are used to effect the blind coordination desired. Individual objects can coordinate with each other by exchanging registrations and group behavior can be effected by allowing a dispatcher to have "global" visibility – again like the traffic signal.

Message- a formal communication sent by one object to another requesting a service. A message may be imperative, informational, or interrogatory in nature.

An *imperative message* asks an object to make some change to itself. No response is expected nor required. The object is assumed to have made the appropriate change. (It is quite possible, of course, for an object to offer confirmation of an imperative message by returning some specific kind of object, but care should be taken that this capability is not misused to create controller objects.)

An *informational message* is similar to an imperative in the sense that no response is expected. It differs from an imperative in that there is also no expectation that the object will do anything at all in response to the message.

An *interrogatory* message is any request for a service. The object always returns an object that encapsulates (embodies) the result requested. The returned object may be simple (a character or a string) or arbitrarily complex.

Messages frequently take the form: **`Receiver Selector (Arguments)`** ← **`returnedObject`** where:

- *Receiver* identifies who is being sent the message. It may be a specific named object (the object Sara), a generic object (aPerson), or the name of a place where an object resides (a variable) and it is the unknown object in that place that actually receives the message.

- Selector identifies the semantics of the message, the essence of the request. Selectors may be simple symbols (like the mathematical operators, +, *, /, etc.) or descriptive phrases (nextObjectInLinePlease).
- *(Arguments)* are objects sent along with the message and are optional. Arguments are used when the receiver of a request for service also expects the requestor to provide some of the information needed to perform that service.
- returnedObject – an arbitrarily complex object containing or representing the result created as a consequence of receiving the message. In the case of imperative and declarative messages, the object returned is "self," i.e., the object that received the message is indicating it is still available to you. (Usually by keeping an active pointer.) If the object changed in any way as a result of receiving the message, the 'self' that is returned is the newly constituted 'self.' On occasion, rarely for most of you of course, the equivalent of a "does not understand" object (an interrupt object, an error trapping object, etc.) is returned as a result of a message sent to the wrong object or when an object is unable (through no fault of its own) to respond to the message.

Interface (Protocol) - the collection of messages and state change notifications that an object promises to respond to or allow registration for, respectively constitutes is interface. The term protocol is usually reserved for that portion of the interface listing the messages that the object is willing to respond to. The syntax of each message in the protocol is specified and is considered the *message signature*. Where arguments are expected along with the message, the signature specifies the class (or type in some languages) of the object expected. The class (or type) of object returned in response to the message is also part of the signature.

The preceding terms are considered essential because they embody everything you need to know about objects to successfully decompose a domain, identify and distribute responsibilities among a

collection of objects, create a taxonomy of objects, and even create scripts to guide objects in the completion of specific tasks.

These terms do not provide sufficient definition for those charged with actually implementing software objects. They provide a specification only. Implementation requires concepts associated with the solution space – a programming language and an implementation platform. Also required, vocabulary for describing the "internals" of objects. A supporting vocabulary addresses these needs.

Supporting Terms

Collaboration and Collaborator – a particular type of object cooperation and the object relied upon for that cooperation. Objects are social and convivial things, constantly exchanging messages with each other and cooperating to complete tasks beyond the capabilities of any single object. One particular form of cooperation is treated a bit differently from all the others – collaboration.

Collaboration occurs when:

1. Object A receives a request for one of its advertised services.

2. While in the process of satisfying that request it needs to ask for a service from object B.

3. Object B is *NOT* an object occupying one of Object A's instance variables; an object occupying a class variable of Object A's class; a temporary variable declared in the method Object A is executing in order to satisfy the original request; or, an object supplied to Object A as an argument of the message requesting the service. (In languages supporting the concept of a "Pool Dictionary" – a

kind of semi-global collection of objects – they too would be excluded from the collaboration relationship.)

Object B becomes the collaborator – a covert assistant to object A.

It is the covert nature of the exchange – it occurs inside the encapsulation barrier of the object that makes this particular exchange different from all other object-object messaging. Collaboration is an aspect of *how* an object satisfies a request. Collaborations always involve some degree of coupling between both parties of the collaboration and the number of collaborations should, therefore, be minimized to the greatest extent possible.

Class- a term with many meanings including: 1) a label for a set of similar objects, 2) an exemplar object, 3) storage location for knowledge (rare) or behavior mechanisms identical for all members of the class, and 4) an object factory.

1. Class as set. An application of the principle of aggregation, noted in chapter **???**, a class is a convenient label for a group of objects that have the same behaviors. Instead of referring to Alice, Dick, Jane, and Bob we can refer to Person meaning any one of those individuals. Objects that are included in the class set are said to be *Instances* of that class.

2. Class as exemplar object. A confusing, for the beginner, but common practice is to use the term class and object interchangeably with the context indicating which is meant. For example, we talk about modeling an object's behavior and we may even visualize a particular object (like

Mary) but we talk about the class, saying, "a Person can identify itself." When we do this we are using the class as an exemplar of the objects it instantiates.

3. Class as storage location. This use of the term is only relevant in the context of software objects. The mechanisms that allow an object to fulfill its responsibilities are, in the case of software objects, blocks of implementation language code. It is more efficient, both in terms of physical storage and, more importantly, in terms of maintenance to store that mechanism in one place - in the class. Instances of the class are given access to the common mechanism when required. In the real world this use of class does not normally occur because storage and maintenance are not usually important issues. All human beings, for example, have bits of neural tissue that, when activated, allow them to respond to the message, "what is your name." The fact that this bit of tissue if duplicated in every human being does not bother us. It is unlikely that we will ever find a more efficient bit of neural tissue and therefore unlikely that we will ever recall all human beings for a "brain update." Conversely, in the case of software we do not want to duplicate even one line of code several billion times, and we are quite often faced with the need to update an algorithm or a bit of code. Therefore it makes sense to store that code (mechanism) in one place. It is also possible to store information shared by all instances of a class in the class itself, and for the same reasons. This use of a class for storage is rare because the object stored in the class must literally be the same one, with the same value and state, appropriate for all instances existing at a particular point in time.

4. Class as object factory. Another metaphor, that states that a class has the responsibility of creating new objects, new instances of the class. To do this it needs a specification for an

instance (an object). When we define a class we co-mingle what are really two definitions, one for the class itself and one for the objects it will be responsible for creating. This will be clarified in the definition of implementation terms, below.

Class Hierarchy (Library) - the taxonomic organization of a group of classes. In some implementation languages (Smalltalk) this is an organized hierarchy while in others (C++) it may be a simple shared library with no hierarchy except compilation dependencies. A class hierarchy should, ideally, function as a kind of index to a potentially large group of objects. For example, we might be looking for an object that collects things and go to the class hierarchy to find Collection Classes. Upon further reflection we might remember that things in the collection must be maintained in a particular order and find the subclass OrderedCollection and SortedCollection. This process could continue until we discovered the class we needed or found that none was available in which case we would create it and add it to the taxonomy (usually at the place where our search ended).

Always remember that the hierarchy is based on behavior and that classes lower in the hierarchy are assumed to extend the behavior of those above them in their branch of the hierarchy.

Abstract/Concrete - labels for classes that do not have instances and those that do, respectively. When creating a taxonomy it is convenient, and sometimes, necessary to create a class solely for the purpose of representing (and in software taxonomies, storing) behavior common to two or more other classes. These classes are not intended to have instances. Concrete classes have instances so, by definition, all objects are instances of concrete classes.

Abstract classes are always parent classes. An abstract class cannot appear below a concrete class in the hierarchy.

In languages lacking explicit taxonomic relationships among classes – an Abstract Class represents a 'template' for a set of related classes. Conceptually it fills the same role – a convenient place for storing specification or implementation material that applies to a group of classes which in turn will have actual instances.

Inheritance - a mechanism enabling objects to access mechanisms stored in their class and any parent classes of their class.

In some languages this mechanism is hidden and generally not accessible to the programmer, in others it must be specified for each program. Languages with implicit inheritance built in have the advantage of being optimized as a result of experience to make programs written in that language as fast and efficient as possible. Languages that expect the developer to explicitly create the inheritance mechanism will be as effective as the developer is skilled. The tradeoff is greater control.

When classes have one and only one parent they are said to participate in a scheme of *single inheritance*. M*ultiple inheritance* implies two or more Parent classes exist for a given Child class. Figure 5.5 shows a fragment of a possible class hierarchy. Person, Student, and Employee illustrate single inheritance while SeminarStudent is an example of multiple inheritance. The circles, aStudent, anEmployee, and aSeminarStudent represent instances of their respective classes. Person is an abstract class and has no instances.

F05xx05

Figure 5.5 – Inheritance Tree Fragment
Shows both single and multiple inheritance

The italicized entries in the classes represent messages that can be sent to objects of that class to obtain an object holding the information implied by the message name. Send the *name* message to aStudent or anEmployee object and you will get back aString representing that objects identification.

Inheritance can be visualized as follows. The *salary* message is sent to anEmployee object. The employee object does not physically contain the mechanism to respond so the inheritance mechanism kicks in and the object is given a copy of the mechanism stored in the class. It uses the mechanism and returns the expected string object.

In another example, aStudent is sent the *name* message. This time the inheritance mechanism does not find the requested mechanism in the Student class so it looks in the parent class where it is found and the student is given access.

Inheritance is intrinsically slow because it must execute machine cycles to conduct its search for the appropriate mechanism. This is why it is important that the inheritance mechanism itself be optimized.

What happens if aSeminarStudent is sent the message *salary*? A long search chain begins with a query to the SeminarStudent class, the Student class, the Person class then the Object class, all with no results. Having reached the top of the hierarchy the search returns to SeminarStudent and then to its

second parent, Employee, where the mechanism is found and made accessible. Multiple inheritance increases the potential length of the search chain and increases the time required to obtain the needed mechanism.

An even more problematic situation arises when aSeminarStudent is sent the message *id*. The needed mechanism is not in the SeminarStudent class so the Student parent is queried and a mechanism of the correct name is found and theSeminarStudent responds with a string representing a student identification number.

Now suppose that it is 3:00 AM and theSeminarStudent is in the vault at the Federal Reserve Bank. The requesting object is a security guard who wants to make sure you are authorized to be in the vault. When theSeminarStudent returns her student id she is likely to be arrested. This situation can occur because classes may have identical message signatures in their protocols. (See the definition of polymorphism, below.)

Multiple inheritance is a bad idea and allowing it is inconsistent with object thinking. It is a bad idea because it needlessly complicates the process of providing objects with access to the mechanisms that allow them to respond to messages. It is a bad idea because it introduces the potential for error and confusion when the wrong mechanism with the correct name is used.

Obviously, there are ways to minimize the negatives that arise when multiple inheritance is allowed. There is no way, however, to elegantly get around the "coupling" between objects participating in a multiple inheritance relationship and the objects requesting services from them. Either

the service provider must be able to differentiate among potential clients and contain logic for directing the inheritance mechanism appropriately, or the client must be aware of the family history of the service provider and know how modify its request for service in an appropriate manner. Or, objects of one class are dependent on the interface defined for another and changing of one requires changing of both. In all cases there is a strong dependency between the service provider and the client that makes both of them vulnerable to changes in the other.

Object thinking will almost always provide alternative solutions to any design problem that, at first, seems to mandate multiple inheritance.

Delegation - a way to extend or restrict the behavior of objects by composition rather than inheritance.

If the need to create a SeminarStudent object arose, it could be done by creating a new class with that name. The new class would inherit from Object (by convention the root of the hierarchy). Contained within the SeminarStudent class would be an instance of the Student class and an instance of the Employee class. The protocol for seminar student would include the protocols of both of those objects as well as any message signatures unique to SeminarStudent itself.

Also included in the protocol would be two messages, *asStudent* and *asEmploye.* When aSeminarStudent receives either of these messages it would assume the indicated role and be able to respond to employee or student messages as appropriate to its current persona. Messages received by theSeminarStudent would actually be routed to the contained anEmployee or aStudent object. That

contained object would then generate the expected response, to be returned in turn to the client of theSeminarStudent.

Delegation absolves the service-providing object of any need to keep track of its clients or the structure and abilities of any other objects. It is a generally a superior solution for the apparent need for multiple inheritance. (It does not necessarily eliminate the interface dependency, but does allow for the possibility of eliminating that kind of dependency. Clients need to be aware that their selected server object has two personalities and request it to adopt the desired personality before sending other messages. But this requires no more knowledge of the service-provider than is known about any other object. The protocol reveals the dual nature and specifies the means to invoke either personality.

Delegation is used extensively in object-like languages like Visual Basic and is a generally useful technique that was too often overlooked when inheritance was being stressed as fundamental to object implementation.

Polymorphism- When I send a message to an object it has complete leeway to interpret that message as it sees fit. The receiver of the message decides the appropriate response and how to respond. As a sender of the message none of that is my concern. Because each object can interpret a message any way it wishes (as long as its protocol tells me what to expect when sending the message), more than one object can respond to an identical message. We have one message that results in different responses. Each response is a 'form' of response and the collection is therefore many (poly) responses or poly forms, or polymorphs – hence polymorphism.

Making the receiver of the message responsible for its interpretation gives me a great deal of freedom in how I work with mixed objects. I can talk with objects in ordinary vernacular - asking them to print themselves, for example - using a single message, "please print yourself," instead of remembering a different message for each kind of printable object. A graphic object in the group will hear the "please print" message and interpret it appropriately for a graphic while a text string will hear the same message and interpret it appropriately for a text string.

Polymorphism is a completely unremarkable phenomenon in the natural world. We expect real world objects to behave in this fashion and make jokes[2] when situations arise when objects respond to messages in unexpected ways because they are polymorphic. Polymorphism is important in software because it relieves me of the burden of coining an infinite number of message variations so that I can satisfy the demand of a language compiler for making every subroutine invocation unique.

In the case of software objects, message signatures are supposed to capture the essence of the message so potential senders will know how to select the appropriate message and object for their purposes. Good names are in limited supply and many objects in quite different areas of the class hierarchy will have similar behavioral needs. Both students and employees have a need to identify themselves so it is not unreasonable to allow them to have identical message signatures to retrieve an identification object. In one case a string might be returned in the other a number. Even if in both cases a string is returned it likely will have a different value.

Encapsulation - the personal integrity of objects should not be violated. This is true whether the object is a software construct or a person. The public-private boundary is assumed to be impermeable. The "insides" of the object are 'encapsulated' - internal structure is hidden and inaccessible.

In most ways encapsulation is a discipline more than a real barrier. Seldom is the integrity of an object protected in any absolute sense, and this is especially true of software objects, so it is up to the user of an object to respect that object's encapsulation. For example, even though it is possible to insert wires into the brain of a human being and, by applying small amounts of electric current, to make that human perform tricks, to do so would be considered a gross violation of the person's integrity. Users of software objects should demonstrate the same respect.

Encapsulation implies more than respecting the public private boundary. Object users should not make assumptions about what is behind the barrier either. This is true whether the assumption is about a piece of knowledge that might be stored in the object or the details of any message response mechanisms that might be inside the object. Just because an object indicates it can provide a bit of information does not necessarily meant that the object has that information stored within itself - it may very well be obtained from some other object without your knowledge.

It is frequently the case that an object will be given responsibilities for maintaining and providing bits of information that they do not possess. As a typical human object, for example, I have a responsibility to remember the names and birth dates of my family members. I do a reasonably good job

of remembering the names because I have memorized them – they are somehow part of my internal memory structure. I do not remember the dates. An external planner object, containing a page object, which lists my family members and important dates associated with each one, is a very valuable collaborator. To an external client, however, it would appear as if I contain both kinds of information because I can respond to requests for both names and dates.

Component - a "large grain" object made up of several basic objects. A roof truss is an example of a component. It is made up of 2x6 pieces of lumber and gang nails. It simplifies the process of constructing applications (in the case of a truss a roof or a house). A component has an interface independent of the interfaces of its members.

Components are frequently known as "business objects" and an example might be a checkbook. A checkBook is composed of a checkRegister, and a collection of numberedChecks.

The distinction between an object, a component, an application, and a system is somewhat arbitrary. All could be called objects because all are packages of behavior with an external interface and all, even a base object, might contain other objects.

Framework - another term with multiple meanings the four commonest being:

- An implementation framework is a collection of classes that collectively capture the behavior of a small, specialized domain. Examples might include a graphics framework or a money framework. At this level the framework consists almost exclusively of a set of classes, a small class hierarchy that can be added to a development environment.

- An abstract or foundational framework is both a collection of classes and the scripts that guide their typical interaction. A foundational framework offers an abstract solution to a particular problem that might be encountered in numerous applications across a variety of domains. Examples of foundational patterns include: object routing and tracking, object allocation and scheduling, object persistence, and many others. An object routing and tracking framework, for example, would be useful in the construction of applications as varied as package tracking software for a parcel service to electronic network management software. Conference center room scheduling, ticket sales, and airline seat management applications could benefit from an object allocation and scheduling framework.
- Application frameworks are customizable solutions to common domain needs. They are also known as vertical market frameworks. Examples would include: an inventory control framework, an accounting framework, a demand deposit framework, etc. This kind of framework is a completely functional software solution, but one that is easily edited (not reprogrammed) into a custom solution for a specific client.
- Architectural frameworks are the most general, and might better be thought of as architectural "patterns." Examples would include client-server, model-view-controller, pipes and filters, blackboards, presentation-abstraction-controller, and many others.

Pattern - another term with at least two definitions. Most of the people using the term were inspired by the work of Christopher Alexander, an architect that proposed a "pattern language" of design parameters that would allow the construction of anything from an "independent region" (Pattern 1) to a montage of photos on the wall of a dwelling ("Things from your life," Pattern 253).

Alexander's goal was the discovery of the principles behind a "Timeless Way of Building," including structural, organizational, and esthetic elements. Alexander's work tends to oscillate between the highly pragmatic and the semi-mystical. It is unsurprising that his work is subject to a wide range of

interpretation. In fact, the intensity of argument regarding what patterns really should be is rivaled only by the original arguments about object programming.

Richard Gabriel, James Coplein, et. al. represent those that believe the semi-mystical aspects of Alexander represent the true essence of his message. For them, a pattern - and even more importantly, a pattern language – is quite different that what has been popularized as a pattern. The popular view, promoted by the "Gang of Four,"[3] (GoF) present patterns as elegant solutions to discrete design problems. Numerous patterns of this sort might be incorporated in any given application.

Patterns for object thinkers are mental shortcuts or cues that direct thinking along known paths and facilitate the discovery of a problem solution. Of course this means that the patterns themselves must be consistent with object thinking philosophy and principles or their use will be counter productive.

Patterns most useful to object thinkers should be derived from the problem domain – just as are objects. They should facilitate thinking about coordination and scripting of objects or useful ways for assembling objects into components or applications. They could be considered "Alexandrian Patterns." Few of the patterns (about 6 of the 23) presented in the GoF book satisfy this demand.

Patterns that reflect the solution space are useful to the object thinker but in a very different way. If you have thought about and designed an object solution to a problem you must still implement your solution using a programming language. Not all languages directly support object ideas and many languages contradict object principles. If you must use such a non-optimal language, then design

patterns of the sort in the GoF book provide insights that can minimize the distortion caused by the implementation language.

GoF patterns might better be called "implementation patterns" than design patterns. Other types of implementation patterns would include the "coding standards" that constitute one of the twelve XP practices. In particular, programming 'idiom' or 'style' provides a very powerful pattern for implementation as well as for communication among developers.

Implementation Terms

Those charged with the actual construction of software objects need vocabulary to support their work – and support their object thinking – while dealing with "the details."

Method - the name given to the block of code that is executed in response to a specific message. Each message in an object's protocol must have a corresponding method.

As noted earlier, at this level it can be difficult to differentiate an object method from a traditional sub-routine. Correct object thinking, however, will be reflected in the method collective. As an aggregate whole, object methods will vary in significant ways from a collection of routines or functions arrived at by applying traditional computer-thinking while programming. For instance, there will be fewer of them and they will be simpler (on average) in their construction. Common control

structures, like CASE-Statements and explicit looping constructs will be absent (or if present, will be very few in number and will be used within an object and not for object coordination or scripting).

Some of the methods used by an object might be considered *private*, meaning that the object itself intends to use those methods and would prefer that they not be invoked by other objects. Public and private are concepts that apply to both methods and the messages that invoke those methods. An object will frequently send messages to itself, resulting in the execution of a private method: to obtain internally stored information or obtain access to an object created in another method. Some languages enforce a distinction between public and private messages and their corresponding methods. Others do not, relying instead on the integrity of programmers to respect the object's design.[4]

Variable - a location or a container where an object may be located. *Class variables* are variables that are part of the Class's permanent structure. *Instance variables* are variables that are part of an object's permanent structure.

Variables have scope, just as they do in any programming language. *Global variables* are visible and accessible by all objects. *Class variables* are visible to the class, any subclass, and all instances of the class and any subclasses. *Class-instance variables* (only exist in some implementation languages) are visible to a class and its instances but are not inherited. *Instance variables* are visible to the object and to instances of all subclasses of the class to which the object belongs. *Temporary or Method variables* are visible only within the method in which they are declared and only as long as the method

itself is operative. *Block variables* are visible only with the block (a special kind of code segment) in which they are declared.[5]

Messages can be sent to a variable, in which case they are received by and responded to by the object residing in that variable location.

Late/Dynamic Binding - This is another programming term that assumes a particular importance in the world of objects. "Type" is a permanent[6] label attached to an object – kind of like a brand or a tattoo - that allows certain objects to be restricted to certain locations (variables) and allows a language compiler to enforce that restriction. In some languages the concept of type is minimized so that it becomes important only when the program is actually executing. Those languages are described as allowing dynamic or late binding. Not all implementation languages support dynamic binding and whether or not they should is a matter of much argument.

The value of dynamic binding is how it allows a developer to take advantage of the natural polymorphism of objects in a direct and intuitive fashion. It is possible, for instance to create a variable, DisplayQue, which will, at various times contain a text object, or a graphic object, or even a movie clip object. I can send the *display* message to the DisplayQue and whatever object is occupying at that point in time will receive the message, interpret it (polymorphism) and display itself. I do not worry about the heterogeneous content of the variable.

A disadvantage arises from the possibility that I accidentally send an object to the DisplayQue that does not know how to display itself, does not respond to the *display* message. This can be

prevented by asking any object seeking entry to the DisplayQue if it understands the display message. If it does it is allowed in, if not it is refused entry.

The drawback of dynamic binding is the possibility that an inappropriate object might come to occupy the variable, receive a message that it does not understand, and cause the program to fail. In “type-safe” languages, erroneous assignment errors of this kind would be detected by the compiler, flagged, and corrected by the programmer before the program is allowed to execute.

Solving this dilemma in a dynamically bound language is straightforward. If the situation requires, I merely ask any object that seeks to occupy a variable if it understands a particular message or if it is an instance of a particular class and grant or deny residency based on its answer. This is extra work of course and may have an impact on performance. Deciding when I need to employ this kind of "type checking" is an important part of design and of object thinking.

The merits of early and late binding are a source of considerable argument. We need not be concerned with that here. It must be noted, however, that object thinking is better reflected in an environment that allows dynamic binding. In the spirit that everything is an object, variables too are objects and should have the responsibility of maintaining their integrity instead of giving that responsibility to another object, like a compiler.

Auxiliary Concepts

The following terms do not define objects or concepts about objects. Instead they add nuances, alters our understanding, or enhances our perspectives of these familiar terms.

Domain - refers to the arbitrarily bounded space we are simulating, in whole or in part, with the objects we design and implement. We understand objects based on how they reflect phenomena and concepts in the domain. A domain might be a business enterprise or a type of business (e.g. banking or government). A domain might be nothing more than a focused community of objects collectively providing a particular set of services; for example, the domain of graphics or the domain of money.

When we are constructing a class library or a set of components our object should be to create a set that is capable of simulating the entire business enterprise or, preferably, the industry of which the enterprise is a member. It is almost always a mistake to define the domain as coextensive with an application program. Yet this is the level at which too many OO texts provide illustrations and examples.

It is possible to define your domain as the computer – the implementation environment. This is, in theory, what object language designers do. If that is your domain of interest, then object thinking should apply equally to that domain as it does to any other. Relatively few attempts have been made to apply object thinking to these domains. Of course, it is precisely this area where you might expect the greatest resistance to object ideas and the greatest allegiance to machine thinking alternatives. Perhaps this is appropriate, but it would be a lot of fun to seriously attempt the creation of an operating system that reflects “pure” objects and nothing but objects.

This is probably a good point to insert a caveat about the material presented in subsequent chapters of this book. Almost all of the examples and discussions assume are focused on application domains. Both an "implementation domain" and an "execution domain" are assumed.

The *implementation domain* will be a programming language and its accompanying class library. In most examples we simply assume the existence of objects like strings, characters, and collections. Also assumed are typical behaviors for those objects. Typical behavior being a superset of those included in t object programming languages like Smalltalk, Visual Basic, C#, Java, and C++. Sometimes, especially when dealing with collections, our assumptions are biased by the capabilities of Smalltalk collections, which are more extensive than in any other language.

An *execution domain* consists of the virtual machine or compiler and the operating system. Again, assumptions are made about services provided by these entities. Also assumed; they are not generally object-oriented environments. Because the operating system, for example, is not object-oriented it is frequently necessary to compromise object principles to some degree in order to make it possible for objects to interact with non-objects.

A special kind of execution domain would be a database management system (DBMS). You could say that the very idea of a DBMS is antithetical to object thinking,[7] but they are an implementation necessity for most organizations. Almost from the inception of object development there has been conflict between object applications and DBMS execution environments. The problem even has its own name – "the impedance mismatch problem."

Business Requirement - defined as any task, decision, procedure, or process that supports a business objective, goal, or mission. A business requirement might be satisfied by an individual object or by a group of cooperating objects. Those objects may be human, mechanical, or software based. If the business requirement can be satisfied by a single object it becomes a responsibility of that object. If a group of objects are required the business requirement is most likely to be expressed as a set of individual object responsibilities plus a script that assures the proper coordinated invocation of those responsibilities.

Business Process Re-engineering – a bridge concept that links object thinking with business thinking. David Taylor illustrates how object thinking can be applied to the business enterprise as a whole, resulting in improved understanding and providing mechanisms for re-thinking the enterprise itself. There is another important link between the ideas of business re-engineering and thinking about objects. The better the object thinking the more likely the need to re-engineer the business so that it matches the simplicity, elegance, and flexibility of the new software. Object thinking almost necessarily reveals better ways to "do business" even when that thinking is ostensibly directed towards software development.

Application - is the term used for a community of objects focused on accomplishing a well defined set of collective responsibilities. As used here the term application has almost the same meaning in object terms as it does in traditional software development.

[1] In this example the second number does assume a kind of passive role, being subsumed by the active number doing the calculation. But either number could have been the active one so the principle of objects acting only on themselves is preserved.

[2] The joke about the large, leather clad, menacing biker accompanied by his equally large and fierce Doberman that enters an elevator car occupied by an middle class couple. When the biker utters the command, "Sit!" the couple as well as the dog immediately drop to the floor. The humor in this story (if there is any) derives from the lack of polymorphism - the couple did not exercise their innate ability to hear a message, determine if it was for them, and react appropriately.

[3] Vlissides, et. al. *Design Patterns*

[4] Languages like C++ and Java also allow for 'protected' messages / methods. This is a way to allow outside objects to send such messages (invoke such methods) but restrict the number of outside objects having that privilege. C++ also allows for "friends" – objects that have access to the private messages / methods of other objects.

[5] Technically, not always true. But this is not a programming text so we will not discuss scoping variations among programming languages.

[6] We are ignoring, for the moment, the possibility of 'casting' types.

[7] Centralized hierarchical control and manipulation of passive data things: clearly, DBMS design and especially relational DBMS design is not predicated on the kind of object philosophy and thinking discussed so far in this book.