7

Discovery

It is said, "a journey of a thousand miles begins with a single step."

Anon

This adage is usually quoted as a motivation, "get started and the rest will follow." The adage misses the potential problem, however, of what happens if that first step is in the wrong direction. It is not necessary to be a fan of chaos theory to recognize that even the smallest change in initial conditions (the first step) can have enormous impact on the eventual results.

For instance, suppose you are in Albuquerque, New Mexico (Bugs Bunny was famous for making a wrong turn here). You want to go to Washington D.C., following a direct line slightly north of east. Unfortunately, you were celebrating your coming departure last night, arose much later than usual and, thinking it was morning and the sun was rising in the East you headed off in the direction of the sun. You might still get to Washington D.C. (the Earth is round after all) but your journey will be rather tortuous and take an inordinate amount of time.

Beginnings are just as critical in software development and initial mistakes are a lot more common than one might suppose. A truism of software development: the most costly mistakes are made the first day. There are two major factors that make beginnings so perilous in software. The first is simply a lack of knowledge. We know the least about the "what," the "how," and the "why" of our development project at the beginning. This should be so obvious that it need not be stated but, surprisingly, we insist on making critical decisions about projects (like overall design, timelines, and costs) at the very time we are least prepared to make them.

The second most common source of error, especially by experienced developers is to anticipate how the computer is going to implement your software before trying to understand how the software should simulate some part of the domain in which it is going to be used. This is "computer thinking" as described in earlier chapters. If you start the development process, as is almost always done, focusing first on the computer you have started your thousand-mile journey with a 180-degree misstep. There may have been a time when computers were so hard to use, limited in capability, and expensive, it was expedient to use them and their requirements as a lens through which the rest of the world was "cut and measured," but that is clearly not the case today.

The argument made by wanting to focus on the computer, rather than simulation of the world, almost always revolves around some kind of issue of efficiency or performance. But, as the developers of SIMULA discovered, if you get the modeling and simulation correct you get a surprising amount of performance as a direct byproduct. One of the axioms of XP admonishes making things work and making them right before attempting to make them fast. Even classical software engineers are adamant about the need to save performance issues to the very end of development. But it is not just performance – it is the focus on the machine as the ground for understanding the software development project that constitutes the wrong first step.

Perhaps the greatest benefit of object thinking is helping your start off in the right direction. It does this by emphasizing the need to *understand the domain first*.

Domain Understanding

Understanding the domain involves three types of mental discipline reinforced by object thinking principles and ideas:

- Extract your understanding of the domain from the domain – use the metaphor of domain anthropology to guide this effort. The goal of software development is to create artifacts, with each artifact consisting of an admixture of hardware, software, processes, and interfaces allowing interaction with other artifacts or with people. Deciding which artifacts are needed and how they will be required to interact with existing (and planned) artifacts, and people, is a result of understanding the domain - as that domain is understood by those inhabiting it.

- Decompose the problem space into behavioral objects and make sure the behaviors assigned to those objects are consistent with user expectations. This requires understanding why users make distinctions among objects and the "illusions" they project on those objects. User illusions, (following Alan Kay), consist of how people recognize different objects in their world and, having recognized an object, what assumptions are made about how to interact with that object and what its responses will be.
- Decompose your problems (applications) in terms of conversations among groups of objects. Everything of interest in the domain is currently accomplished by groups of objects (people and things). Any artifact you construct must participate, in a "natural" way in these same groups. Perhaps your artifact is simply replacing an existing object in the domain with a computer-based simulacrum, in which case it must know how to respond to and supply existing interaction cues. Perhaps it is an entirely new object, in which case it will need to be "socialized" to conform to the existing community.
- Model your objects as simply as possible consistent with the needs of the development team (which should include users, ala XP) to communicate their collective understanding of the objects and the domain. Only after you have completed this kind of modeling should you even begin to think about how to implement any new or replacement objects your domain understanding leads you to believe desirable.

In this chapter we will discuss how object thinking influences your understanding of the domain, its objects, and the models useful to clarify and communicate your understanding. In Chapter 8, Transition to Design, we will outline how to move towards implementation without abandoning object thinking or our understanding of the domain.

Domain understanding involves using another metaphor, domain anthropology, to provide the mental perspective used when discovering objects and their interactions. The same perspective is useful for identifying and assigning discrete services to specific objects. A number of heuristics can be employed to facilitate object discovery and definition. We will now discuss each of these factors in turn.

Domain Anthropology

The software developer must confront a strange world with the goal of understanding that world in its own terms. The closest approximation of this task is the work of a cultural anthropologist confronting a strange (to her or him) society. Domain anthropology provides a very useful metaphor – providing both cautions and insights that will guide the application of object thinking to the early phases (requirements definition, object identification) of development.

Beginning with a caution: anthropologists have their own culture – with its associated explanations, values, assumptions, etc. – and must zealously guard against the mistake of interpreting everything in the new society in terms of their own (often unconscious) culture and values.

Most developers are steeped in the "way of the computer" with abundant assumptions about how a computer works, and the best ways to make a computer perform its tricks. You could say that developers have internalized a culture of computing that they must set aside (just as the anthropologist sets aside their birth culture) before they attempt to understand users and user domains.

A developer attempting to understand a new application or problem domain is not unlike an anthropologist confronting a new culture for the first time. She is surrounded with experts in that domain but they speak and act in "strange ways" that she needs to understand. You need to learn from them how they view and make sense of their world.

As an anthropologist, and an "outsider," you do have a perspective that the domain experts lack. Much of what the domain experts actually do and think about is "invisible" to them in the same what that a human is rarely aware of the culture to which he or she belongs. You will be able to observe important aspects of the domain that the experts will fail to report to you or talk about because they are "so obvious" that they are below the threshold of consciousness. But do not mistake this kind of objectivity with projections of your own cultural values (computer knowledge) onto the domain you are supposed to be observing.

When a cultural anthropologist confronts a new culture the first task (after allaying suspicion and obtaining some kind of rapport[1]) is to learn the language. The object developer needs to learn the names of things in the domain that have been accorded object status – made distinct from the domain and from other things in the domain - by virtue of having been given a name. Naming things is how domain experts, (and the indigenous peoples studied by the cultural anthropologist), naturally decompose their domain.

Most books and methods addressing how to do object development recommend that the object discovery process begin with underlining the nouns (names) in a domain or problem description. While it is true that many of those nouns will indeed turn out to be viable objects it is unlikely that any written description will be sufficiently complete or accurate to meet the needs of domain anthropology. Cultural anthropologists do not base their understanding of a culture on written reports by people in that culture. They go and live with the people, observe them, and talk with them. Domain anthropologists must do the same thing.

Domain Anthropology and Management

Anthropological investigation of a culture is not an easy task. The anthropologist will spend months, if not years, preparing to "go to the field." That preparation includes digesting all that is already known about their target culture (and other similar cultures) as well as techniques for observation and information elicitation. Once in the field, the anthropologist will spend a minimum of a year, optimally two years, living and interacting with the people he is studying.

A domain anthropologist should do no less! Observation for a complete business cycle (including the annual financial cycle) that includes the major events and rituals (new employee to company picnic) that occur in the domain will yield important information relevant to the design and success of any new software system.

However, management is unlikely to provide the necessary time for this kind of activity – at least not as overhead for any single project. Therefore it will be necessary to define shortcuts and means of sharing domain knowledge among different projects. Shortcuts include: on-site customer (rely on the customers existing domain knowledge instead of trying to acquire it yourself), joint application development (JAD) techniques, and as much focused observation you can get away with.

Given that management will likely not allow time for a complete ethnography, some kind of conversation about the domain with both users and developers participating is desirable. Constructing a semantic net (see Chapter 6 – Method and Models) is a useful way to gather information about potential objects and expectations of those objects. While constructing this net, the domain anthropologist must operate, somewhat, as if she were a naive child – constantly asking:

"What is this?"

"Why is this considered to be a ______?"

"How do you distinguish between this and that?"

"Why?"

The answers to these questions not only reveal potential objects, it also starts to reveal the expectations that domain experts have of those objects.

F07xx01

Figure 7.1 – Semantic Net
A partial semantic net for the subsidized mortgage company

Begin the object discovery via semantic net technique by asking for one or two sentences about the domain in general – e.g. two sentences that describe what your company/department/team does. Select a couple of nouns and verb phrases and write the first elements of a semantic net (Figure 7.1) on a whiteboard. Ask everyone present to brainstorm new elements (circles, nouns) and new connections (arcs, verb phrases). This exercise should be a stream of consciousness – brainstorming – session with analytical discussion to follow later.

It usually takes but a short period of time, five to ten minutes - to produce a fairly large diagram. The circled nouns provide a rich set of potential objects and the relationships an equally rich set of potential responsibilities (expectations of those objects). This semantic net can then be placed on the wall as a common reference point as the discussion proceeds to more detailed levels.

Subsidized Mortgage Company

The partial semantic web in Figure 7.1 is based on the following conversation about a company that provides subsidized mortgages to qualified couples.

Please give me a brief description of your company.

Our company provides mortgages to qualified couples, subsidizes monthly mortgage payments when necessary, collects payments, invests whatever capital is not tied up in mortgages, and otherwise manages its assets. [This yields an initial net consisting of bubbles for: mortgage, couple, payment, investment, and asset. Initial arcs include: investment is-an asset, couple receives mortgage.]

I assume the mortgage is for a home?

Yes, a home that must have a purchase price less than or equal to the median price of all homes in its neighborhood. Oh, and the home cannot have a purchase price greater than $100,000.

What qualifies a couple for a mortgage?

Couples must have been married for at least one year, but not more than ten, and be gainfully employed. They will not have the required ten-percent down payment required by other lenders, and the monthly payments on a $90,000 mortgage will exceed 28% of their joint gross income.

What do you mean by gainful employment?

At the time of application proof must be submitted of full-time employment for at least 48 of the previous 52 weeks.

We also have to have funds available sufficient to cover the mortgage.

How do you know if you have funds available?

We calculate our expected income from investments divided by 52 (we calculate available funds weekly); our annual operating expenses divided by 52: weekly income from mortgage payments (oh yes, we also collect mortgage payments weekly); and any grants we expect to give out this week. Expected annual income from investments is calculated and divided by 52 and added to the pool of funds. We then compare the cost of the mortgage, or mortgages, applied for with the pool and fund all we can. Any left over money is invested.

People have to apply for the mortgage? [Note the naïve question.]

Of course, they fill out an application and when the information on the application is verified the application is deemed fundable. If we have money, we grant the mortgage and the application is given funded status. If we do not have the money this week, the application stays fundable until it is funded or some other event causes it to be cancelled.

One point of reducing the object discovery process to simple questions and observations is to eliminate preconceptions. If a cultural anthropologist were to begin an analysis of kinship with the preconception that everyone lives in a nuclear family, (mom, dad, and 2.5 kids), and everyone practices monogamy their analysis of human kinship patterns will be so incomplete and distorted it will be worthless.

An analogous result will occur if the domain anthropologist begins her task with preconceptions about data, functions, processes, " how things must be implemented in COBOL," or "how computers work." These preconceptions about implementation will blind the domain anthropologist to what is going on in the domain and what is required of the new software artifact. For instance, a software expert is likely to see any kind of form as an ordered collection of static text objects and entry field objects because that is the way that a form will be implemented. If this knowledge disposes them to ask users only those questions dealing with what text, what entry fields, and what values can be placed in an entry field - they will miss potentially important behavioral requirements. Behavioral requirements for a form might include: identify itself, modify itself, present itself (with the implication of multiple views), complete itself (make sure every entry field contains appropriate values), and validate itself. These are behaviors that will not be discovered by asking about contents.

Another warning. We have “trained” users over the years to anticipate how software developers think. It is not unlikely therefore that they, at least initially, will try to tell you what they think you want to hear. They will couch their responses to your questions in terms of data, attributes, algorithms, and functions. When this happens they must be questioned further to discover the behavior involved.

The metaphor of an object as a person is important for developers to begin to understand objects. It is equally useful for domain experts and should be discussed with them preliminary to any question and answer session. Both the developer and the domain expert can then use a common language in their discovery dialog. For example:

“Well, there are customers” (User offers a potential object)

“What do you expect of a customer?”

“It should tell me its name, or identify itself.”

“What else”

“It has a social security number.”

“Do you want it to tell you its social security number?”

"Yes."

"Then we can say it must report or tell you its social security number."

"Yes, and it should tell me its address, phone number, age, gender, birth date, ..."

"Wait a second, you mean it should be able to describe itself and tell you how to contact it?"

"Yes, yes exactly."

To this point domain anthropology has focused on identifying some potential objects on decomposing the domain. It is equally important to discover the "social relationships" among those potential objects. Human cultures have rules for interaction among its members. People engage in a complex web of mutual obligations that change with time and circumstance. People participate in relations, some fixed (like biological parentage) and others temporary (like employer-employee). Similar social relationships exist among objects and they too must be discovered.

Details, and in some cases discovery, of some of these relationships will be deferred until the focus of development is on the transition to design. The domain anthropologist focuses on how and why objects interact. In some cases, why they do not interact – i.e. what rules prevent a specific object from certain types of interactions with other objects.

How objects interact is mostly a matter of recording actual and possible (desirable) conversations among objects. Why is mostly answered by seeking the task(s) the motivated the conversation. There are different ways to organize the work required to solicit all the conversations and tasks relevant to our understanding of the domain and of the objects we might have to build to operate in that domain.

One way – fairly traditional in a software engineering sense – would be to isolate a subset of the objects in the domain, draw an arbitrary boundary around them and label the collective as the "NewSystem" object. You could then ask each remaining object in the domain to identify what services it might require of the NewSystem object. The set of non-duplicated services constitutes the responsibilities/behaviors of NewSystem. These, in turn, will need to be examined to expose the actual multi-object conversations required to provide the identified service. Conversations might have sub-conversations. Eventually this process will yield a set of nested conversations – each nested thread being a "use-case" as that concept was developed by Ivar Jacobson and incorporated in UML.

Each threaded conversation can be modeled, using UML syntax, with a high level diagram (similar to the context diagram of structured development) that represents each service consumer (user role) as a labeled stick figure, and the NewSystem object as a labeled circle - see Figure 7.2. The making of a request by a service consumer defines a pre-condition for some conversation among objects within the NewSystem circle. The return of an object to the service consumer constitutes a post-condition for that same conversation. The conversation itself can be modeled with an interaction diagram (see syntax definition in Chapter 5, Method and Models.) Interactions diagrams can be nested – although UML graphical syntax provides no way to represent the presence or absence of a sub-scenario. The extended UML syntax shown in Chapter 5 does provide for branching and looping sub-scenario notation for the interaction diagram.

F07xx02

Figure 7.2 – Partial Use-Case (context level and interaction diagram)
A very partial example of the major parts of a UML use-case.

This same information about the objects in a domain and their interaction can be modeled textually. The System Request Table (Figure 7.3) replaces the stick figure diagram of UML and the Conversation Table replaces the graphical interaction diagram (Figure 7.4).

F07xx03

Figure 7.3 – System Request Table
Service requester is identified at top of table (NewSystem assumed as service provider. Each line identifies the service requested – message sent to NewSystem, the object returned as a result of that request, and notes for explanations if needed. Each line of this table constitutes a pre/post condition for a conversation.

F07xx04

Figure 7.4 – Conversation Table
Pre and post conditions are noted at top of table. Each line of the table shows who is making a request, the nature of the request (message), the service provider, the object returned and a comment if necessary. Conversational order is assumed to be top down.

Use whichever syntax (graphical or textual) you prefer. Remember, object philosophy questions the utility of any representation or model beyond its immediate assistance to those involved in making the model. Don't get caught up in trying to make your model "complete," "accurate," or "true."

There is nothing necessarily wrong – or non-object thinking – with the systematic approach just outlined. To be systematic, however, it presumes a concerted effort to identify all use cases up-front and based on defining the requirements for NewSystem. An alternative approach, one much more consistent with XP ideas and techniques, would be to define stories one object (or a small number of objects) at a time. All the same elements of discovery are present as would be in a systematic approach – it is simply a philosophic preference to do things in the small.

We begin, this time, by looking at the semantic net constructed earlier and noting the relationship (completes) between couple and application. We then write a simple story about this relationship:

> *A couple requests an application for mortgage and fills in all necessary blanks.*

This story gets expanded into a sequence of interactions:

1. The couple glances at the application to make sure it is the right one (implicitly asking the application to identify itself.)
2. The couple looks for the first blank spot (implicitly asking the application for the next space needed to be filled-in).
3. The couple asks what information needs to be placed in that spot (implicitly asking the blank space for its label on the assumption that the label will describe the information needed).
4. The couple enters a value in the blank spot.
5. Steps 2-4 are repeated (next blank spot replacing first blank spot in the description of step 2) until there are no more blank spots.

This conversation can be graphically depicted with an interaction diagram, Figure 7.5.

F07xx05

Figure 7.5 – Interaction diagram
This diagram depicts the user-driven conversation required to fill in a form.

This conversation assumes an "active" couple object and a "passive" form object. In one sense this reflects reality – a paper form is pretty passive and a couple is comprised of live human beings capable of action – but it is not as reflective of object thinking as it might be. Consider the alternative story card:

> *A couple requests an application. The application, couple and entryField objects collaborate to make sure all entryFields contain a value.*

This leads to a conversation with the following steps (graphically depicted in figure 7.6):

1. Couple requests an application from the SubsidizedMortgageCompany (our conversation reveals a new object capable of providing an application).

2. The application presents itself to the couple. (Implicit identity verification plus ability to display itself).

3. The application identifies the next entryField requiring content. (Asks itself for the next field.) and then notifies that entryField that it now has the opportunity to obtain content.

4. The entryField asks the couple for input. (Implicit request, manifest by asking a blinkingCursor to display itself in the area of the entryField, or highlighting itself in some other way.)

5. The couple provides a value.

6. The entryField notifies the application that it is satisfied.

7. Steps 3-6 are repeated until the supply of entryFields associated with the application are exhausted.

F07xx06

Figure 7.6 - Interaction Diagram
Depicts the alternate scenario for filling in an application.

Differences in the conversation are minimal, but they reflect the behavioral parity of all objects. Each object is an active entity in control of its own destiny. The alternative way of thinking about the conversation also reveals objects that might not have been discovered or thought about in the original model of the conversation.

More importantly, the thinking behind the alternative conversational model is more general – more abstract – and therefore applicable regardless of the means of implementing the application. Behavioral expectations of the application and its parts are consistent whether implemented with paper or as a software application. True, the software versions of the objects have more interesting or dynamic ways of exhibiting behavior. The entryField object, for example, using the services of a blinkingCursor instead of simply displaying itself as a blank space.

The alternative model is also more consistent with user expectations of the objects involved – which is the real goal of domain anthropology. If we are to build digital versions of forms and entry fields then they should reflect the expectations of their naturally occurring counterparts. For example:

- We expect application forms (any form for that matter) to "present" their constituent parts (labels and entry fields) to us in order. In the digital manifestation of the form this is accomplished via interaction with a cursor, or more crudely, by presenting the entry fields to the user one at a time. In the paper implementation, the form still presents the elements to us in order, but does so implicitly – taking advantage of cultural conventions[2] – by ordering their placement starting in the upper left and proceeding right then down.
- We expect entry fields to ask us to enter some value. The digital version by using a blinking cursor assistant object to remind us it is waiting and refusing to go away until it is satisfied. The paper version by nagging us simply by remaining blank. "Nature abhors a vacuum" no less than a human abhors a blank spot on a form.

The value of applying object thinking to conversation discovery and modeling will become more evident as we consider responsibility assignment and discovery in more detail. A preview: allowing entryField objects and form objects to validate themselves in collaboration with another object (a rule object). A paper based entry field seeks validation by displaying its content to a human and asking for confirmation or replacement of its displayed contents. A digital version could do the same thing by asking a rule (or rules) to evaluate themselves (to true or false) using the value held by the entry field as one of the rules variables.

A reader note earlier in this chapter suggested that a domain anthropologist brings to bear a certain sensitivity or objectivity by virtue of being an "outsider." Seeing the implicit behavior of forms and entry fields, just noted in immediately preceding paragraphs, is an example of applying this kind of objectivity. It would not occur to a typical user to volunteer how they interact with forms and entry fields, because that interaction is based on cultural conventions that are non-consciously applied. They are nevertheless real and need to be identified and exploited in order to develop and design the most robust objects and applications.

Object Definition

Semantic nets and stories (scenarios, use-cases, interaction diagrams) provide input, but the most critical aspect of discovery is object definition. Each object must be identified in terms of its actual or intended use in the problem space – the domain. The metaphor of domain anthropology continues to shape our thinking about each individual object and what we expect of it in terms of services (behaviors, responsibilities).

Each circle on the semantic net is a potential object. We can select any one of them and attempt to identify all of the services such an object might provide in its domain – *domain, not just the application or specific problem where we first encountered this object*. Typically we would use the first two sides of the object cube model to record our thoughts about the object and its behaviors.

Our goal during discovery is object definition – not object specification. Definition means we want to capture how the object is defined by those using it, what they think of that object, what they expect of it when they use it, and the extent to which it is similar and different from other objects in the domain. Specification will come later, when we allow ourselves to consider how this object might be implemented (simulated) in software. Specification will involve making some design decisions and capturing the information necessary for someone to actually build the software version of the object.

Object definition involves capturing three bits of information: a short prose description (in the words of a domain user) of the object; an enumeration of the services it is expected to provide (and, if other objects are used as helpers for any given service, the kind of object to be used); and, when appropriate, a stereotype – another object(s) which the one we are working with resembles in terms of similarity of services provided.

Side two of the object cube is used to record the description and any stereotype, side one is used to record the list of responsibilities and collaborators. To this point we are engaging in exactly the same thinking and recording process advocated by Beck and Cunningham and their CRC cards. It would not be unusual to actually use 3x5 index cards for discovery as they are cheap, easy to modify, sharable (you can pass them around), and provide a good visual reminder that well defined objects have limited responsibilities.

Where to start? Pick an object or two and a story in which they appeared. This will provide a starting point. For example: Application and entryField. We ask the user (domain expert) for a definition and the user responds, "an application is a kind of form, used to collect specific facts about the people applying for a mortgage and the home they want to purchase." We record this information on side two of the object cube (or, the back of the 3x5 card). The definition, conveniently, provides us with insight into a stereotype for the application object – it is a "kind of form." So we record that fact on side two as well. (Figure 7.7)

F07xx07

Figure 7.7 – Side two of object cube for application and entryField objects
A simple record of the description and stereotype obtained from the domain expert (user).

We do the same thing for the entryField object, obtaining the information shown in Figure 7.7.

We now turn our attention to the services we expect from our application and entryField objects. We have several choices on how to think about the services. We could look at the arcs connecting the application circle to others on the semantic net so see if they suggest services expected of the application. We could go over the story about filling out the application to see what was expected of the application in that story. Either of these options are likely to expose services that our application must satisfy, but those services might be couched in very specific language reflective of the limited context in which we discover the service rather than the domain as a whole. We will take care to generalize any such discovery so that the service reflects the needs of all of the objects that might use our application.

A third way to approach the enumeration of services is to simply ask the question, "I am an application; what services can I provide to others?" To help us answer this question we can look at an actual physical example of the application form and we can think about all the stories involving our application.

We notice that the application has a funny string of characters at the bottom (something like AF001rev10/03) and we also notice a line of print at the top in larger font size than the other contents. We surmise that these character strings identify the form in some way. We also recall that the stories told about the application implied that users of the application looked at it to confirm they had the right form. From this we conclude that the form has a responsibility to "identify itself." We record this information on side one.

Visual inspection reveals several examples of short text strings and several examples of blank spaces. We might give these things names – textString and entryField, respectively – and then create a new object cube with those names on them. We will want to explore those objects later to determine what their responsibilities might be.

Our attention is on the application, however, so we wonder whether the application has any responsibility in connection with these newly discovered objects. It is obvious that the application holds these elements in an organized manner. We might conclude a responsibility that the application holds elements. We reject the phrasing, "holds elements" because that does not describe a service, instead it describes something the application must do in order to provide some other service. A bit of thought, and we come up with the responsibility, "provide access to elements." Implicit within this responsibility are variations like: "provide access to a subset of elements," or "provide sequential access to elements." We might record these variations on side one, so we do not forget them, even though we are not sure they are appropriate responsibilities.

We can see the application, so we know it can display itself. We would record display self on side one. It is necessary to think a bit about what is involved in displaying the application because more than one object is visible – the application of course, but also the textStrings and the entryFields. It is inappropriate for one object to assume responsibility for the actions or behaviors of others. The textStrings and entryFields must display themselves. The application might be the recipient of the display request but it must be careful to delegate some of the display behavior to its elements. We can capture this information by recording that textString and entryField are collaborators used by the application to make sure everything is displayed.

Usually we only want to record one collaborator on a line. To accomplish this heuristic goal, we might stereotype both textString and entryField as displayableObjects. We could then record displayableObject as the collaborator. It would be nice to be a bit more specific about displayable objects we are collaborating with, so we might refine our label (which identifies a new class to be thought about) to displayableElement instead. This implies that any elements the application contains that are displayable are the ones we collaborate with to complete the application's display responsibility.

The fact that the application is a container of other objects also suggests to us a second stereotype for application. It has behaviors similar to a collection – another type of container object. We should add this observation to side two of our card.

There is one more responsibility that needs to be considered. In one of our stories it was suggested that the application form needs to be verified or validated. The information entered on the form – the information contained within the entryField objects – must be confirmed. This would suggest a responsibility of, "validate application form." But, who should this responsibility be assigned to? According to the precepts of object thinking, the application must be responsible for validating itself!

So we add one more responsibility to side one of the object cube, "validate self." Exactly how this is done we do not know. But a bit of reflection indicates that part of the validation is done on the values stored in each entryField. The application should not be validating the entryField, the entryField must be responsible for that itself. Just as was done with the display responsibility, the application will delegate to each entryField its share of the validation workload. We list entryField as a collaborator for the 'validate self' responsibility of the application.

Alas, our thinking may be leading us astray a bit in our consideration of validation. Perhaps it is not one task but two similar tasks, one of which belongs to the application and the other to the entryField. To be sure we are doing the right thing, we need to ask what is meant by "validation" as that term is used in the domain. So we might ask a user, "how is an application validated?" The user then provides a simple story about validation:

> *"Well, the data in each field is checked to see if it the right kind, a name, a date, a dollar figure, things like that. Then we verify that the value is accurate and permissible. By permissible we mean things like the number entered in age has to be between 25 and 40, or the number of years married must be between 2 and 10. By accurate, we mean that the income*

figure is verified with the employer. Also the bank balances, things like that. Finally, we check the application for consistency. Say they enter a city name and a zip code – we check to see that the zip code matches the city."

Thinking about the story suggests that the application, and each entryField, is validated by applying a rule to the value stored in an entryField. In the case of the application, the rule might have to look at values stored in more than one entryField. It also suggests that both the application and each entryField collaborate with a validationRule. So we change both object cubes to reflect the newly identified collaborator. We would also make a new object cube for the validationRule object.

When we are done we would probably have the object cube – sides one and two – for application and for entryField completed as shown in figure 7.8. Once we complete side one and two of an object cube for every object identified on our semantic net, and once we have assured ourselves we can tell any story the user can tell us about using those objects, we are done. The overall result for the subsidized mortgage company will be similar to Figure 7.9 (side one) and 7.10 (side two).

F07xx08

Figure 7.8 Object Cubes
Sides one and two for application and entryField object cube.

F07xx09

Figure 7.9 Object Cubes – side one
Objects identified in the subsidized mortgage example.

F07xx10

Figure 7.10 Object Cubes – side two
Objects identified in the subsidized mortgage example.

As you look at the figures, do not be alarmed to see some classes not directly discussed in the example so far. Once we have identified all objects we apply further considerations, using the stereotypes recorded for each object, to see if it is really an independent kind of object, or an instance of a more general object with the same responsibilities. We will also look to see if we can identify a taxonomy of objects leading to the discovery of a class hierarchy. This effort will be discussed in Chapter 9 – Cleaning Up.

Another Example

Some objects from an air traffic control system (ATC).

Imagine we are working on modeling the world of air traffic control. We will have identified a number of objects in that realm including passengers, airplanes, control towers, flight routes, controlled airspaces, and many more. We will also have captured a number of stories about how objects interact in this world. The following is a very brief illustration of how a few objects might have been modeled in such an example. We will focus on the airplane and some objects that come to be identified as we consider the airplane and its responsibilities. We will use the story of how an airplane fulfills one of its responsibilities – reporting its location as the background for our process of discovery.

Figure SB-7.1 includes a depiction of side one of the object cube for an airplane. By id self we mean the plane can provide is registration number. Describe self entails giving a requester of that service an object containing the characteristics of the airplane (type, manufacturer, seating capacity, etc.) as a kind of keyed array. Report location means the airplane will provide a location object containing all the pertinent information for a location (altitude, latitude, longitude, vector, etc.). Move to new location means the airplane will actually relocate itself in space.

Reporting a location is a lot of work. "First I find my altimeter and ask it for an altitude, then the compass to ask it which direction I am heading in, then the airspeed indicator for my speed, and then the GPSS for my xy (latitude-longitude) coordinates." Being lazy (an admirable quality of all objects), the airplane looks to see if it can delegate some of the work to another object. For example, someone that keeps track of all the instruments. A new object cube for instrumentCluster is created and we list that object as a collaborator on the airplane's cube.

We also discover that the instrumentCluster is required to add and delete instruments from time to time – when they fail or newer versions are available. So we add that kind of responsibility to the instrumentCluster cube. The instrumentCluster's part of the location reporting task is worded as assemble location. The wording of the responsibility suggests a collaboration between the object doing the assembly and the objects having the information that needs to be assembled – each instrument.

Fsb07xx01

Figure SB-7.1 – Object cube side one
Side one for airplane, instrumentCluster, and instrument objects.

This yields three cards with responsibilities as shown in Figure SB-7.1. Figure SB-7.2 is an interaction diagram illustrating the following story about an airplane reporting is current location.

The control tower asks the airplane for its location. The airplane immediately asks the instrument cluster to assemble a location. The instrument cluster asks each instrument in turn for its value, then asks a location object to hold that value. Once all the values have been obtained, the instrument cluster gives the airplane the location object. The airplane then asks itself for its id, asks the location object to append that id to itself, then returns the completed location object to the control tower.

Fsb07xx02

Figure SB-7.2 Interaction diagram
Object communications required for an airplane to tell the control tower where it is currently located.

Implicit, but perhaps not obvious in the example above (and in the sidebar) are a number of heuristics for discovering and assigning object responsibilities.. There are also some points where the developers thought processes might have proceeded in a different direction – there are different and equally valid object designs. Some hints of why the examples shown are as they are can be incorporated in the discussion of the heuristics.

Heuristic: Let object's assume responsibility for tasks that are wholly or completely delegated to other objects in cases when it reflects natural communication patterns in the domain.

For example, the question might arise as to why the control tower asks the airplane for a location instead of the instrument cluster. After all it is the instrument cluster that actually does the work to obtain a location. If the reader is familiar with modern aviation you know that control towers do have the ability to talk to at least one instrument carried by an airplane - the transponder. This question raises a general issue of when an object should or could assume responsibility for work actually done by others.

In the case of the Airplane it seemed a natural call. After all the control tower already has to talk to the planes on all manners of business so it does not seem illogical for it to do so for location as well. The plane is already "visible" to the control tower and the instrument cluster is a part-of the plane. It also does no harm to let the plane front for the instrument cluster and it does a lot of good.

If we decide at some future time to let the control tower talk to the instrument cluster directly all we have to do is update the name of the location source, which could be stored as an instance variable in the control tower object, and the control tower sends the message to this new service providerr. No changes are required in any object definitions. Polymorphism even allows the same message, "location please," to be sent to any potential server.

Heuristic: Use "middle management objects" to get a better distribution of responsibilities and increase reusability.

An example of this heuristic arises from the answer to the question of why the Airplane needs the instrument cluster. Layers of "middle management" are inefficient so why not eliminate them? An answer to this question becomes apparent if we enumerate all the responsibilities that are involved with keeping track of and using a set of instruments..

1. Know which instruments are included in the set.

2. Assemble a location by asking the correct instruments, in the correct order for their values.

3. Remove an instrument from the set if it is bad.
4. Keep track of which instruments belong in which kind of airplane.
5. Add new instruments when available.
6. Monitor the status of all the instruments.
7. Replace an instrument with an improved model when asked.
8. Make one or more instruments available when asked.

This seems like a lot of work for an airplane to undertake, especially when we would like the airplane to concentrate on moving safely from one place to another. It will also increase reusability of both the airplane and the instrument cluster objects by keeping separable responsibilities separate.

When an object accepts responsibility for work that it cannot do alone it does so to meet expectations expressed in its environment. Thinking about the nature of the work - anticipating what kinds of tasks will need to be performed can suggest other objects that can assume one or more of those tasks. This in turn leads to finer grained decomposition - by identifying additional service providing objects with more specific behaviors.

Heuristic: Use anthropomorphization and "foreshadowing" (thinking ahead to object development tasks that will be done at a later stage) to determine if an object should or should not assume a given responsibility.

Experience doing object development allows you to anticipate future steps in the development process and use your knowledge about what you will do then as a guiding heuristic for earlier steps.

This is also part of the iterative nature of object development. Some of your iterations will be done only inside of your head as you think, "if I give this object this responsibility then later I will find that it requires this and this piece of information and will have that kind of method." You can then decide that the responsibility should not, in whole or part, be assigned to the object under consideration. These decisions will not, of course be final, they are just a way to help you complete the responsibility distribution that is your immediate goal.

Anthropomorphization assists us in thinking about responsibilities and other objects in several ways. We can think of objects as being "lazy" and only wanting to do a limited and closely related set of activities. We find the responsibilities that need to be delegated to other objects by foreshadowing what might be involved in fulfilling a responsibility. We can think about how much the object might need to know, if it will need to know the "wrong kinds of things" about other objects (i.e. anything not evident in the object's interface), as well as what it might need to do. Foreshadowing is done as part of analysis, but when you make the transition to design you will address these questions in a systematic fashion.

Heuristic: Responsibilities should be distributed among the community of objects in a balanced (no object should have a disproportionate share of the responsibilities) manner.

How is it determined if an object has too many or too few responsibilities? If you use a three-by-five lined index card to record you responsibilities - one to a line - you will find you are limited to six or seven entries. This is an excellent heuristic for the maximum number of responsibilities that an object can or should assume. The lower bound is an object must have at least one unique responsibility or there is no justification for making it a separate kind of object.

A balanced distribution of responsibilities across a group of objects is the goal. Also implied is a balance in the distribution of knowledge among those objects. An object should not do to much nor should it know to much. You can determine the knowledge distribution, roughly, via foreshadowing as noted above.

Heuristic: Always state responsibilities in an active voice describing a service to be performed.

For example: a common tendency for new object developers is list as responsibilities things that an object must “know.” An airplane must, “know its current location.” While this may be true it can be misleading. It implies too much about how the object may be implemented because “knows” implies an instance variable. It is also quite possible that an object will know things that it will not be willing to share with others and therefore will not be included in the interface for that object. Always state your responsibilities in terms of a service, with an awareness of a possible client for that service.

Heuristic: Avoid responsibilities that are “characteristic specific,” that focus on providing a potential user with the value of a single characteristic of the object.

It would be a mistake to list responsibilities like: know age, know height, know eye color, know weight, (see preceding heuristic). It is also a mistake to list: report age, report height, report eye color, report weight. Even though the latter are in an active voice they are characteristic specific and should be avoided. The source of the problem is that an object may have a lot of characteristics and you only have room for six responsibilities. It is better look for an umbrella label for the responsibility, like "describe yourself," or "identify yourself." Important advantages will be gained by following this heuristic.

One advantage is flexibility. If you make the interface to your class dependent on the characteristics it may need to report and someone decides on a new characteristic you will need to redefine your class. The alternative is to have a responsibility of, "describe yourself," which results in giving the requestor a description object. This description object is a collection of values for the objects characteristics. It can be asked for one or more of the values it contains so the client gets the information it needs. However, when you decide to add a new characteristic to an object you can do so by simply sending a message to the object telling it to add this new trait to its description. No redefinition or recompilation is required.

Another advantage arises from the fact that different clients need to ask about different characteristics. If the characteristics are defined in the interface a situation requiring multiple object types (classes) is created. For example, it is likely that a credit department will need to know characteristics of customers that are not shared by other departments in a company. The temptation, when characteristics are part of the interface (or when using a data driven approach to design) is to create a small class hierarchy with Customer, Cash Customer, and Credit Customer.

If the characteristics of a customer are held in a description object there is a need for only one Customer class. All clients can ask the customer for its description and then ask the description for those characteristic values in which it is interested. Cash or credit become just additional characteristics of the object.

There are some obvious exceptions to this heuristic. Almost all objects will need to identify themselves on occasion. An identity can be a characteristic of the object just as a social security number, (or, for some, a prominent tattoo), is a characteristic of a human being.

Deciding if a characteristic merits a separate responsibility will be revealed in the collection of scenarios in which the object is involved. If an object has ten characteristics, for instance, and one of them is asked about frequently and by a number of clients then it should probably be a separate responsibility. If the other nine are the subject of only occasional inquiries then they should be grouped in some fashion and the responsibility becomes to return the group.

Heuristic: Create proxies for objects outside your domain that are sources of information required by objects within your domain.

In the process of completing both the binary use-cases and the scenarios a number of objects that are outside your domain will be identified. Legacy systems and human beings are the most obvious examples. They continue to be of interest and importance because they are sources of information or behavior that your objects will require. These requirements will be exposed in the scenarios as messages sent to these outside objects.

For instance if I was an order object I might need to obtain information and decisions from a human being. I will want them to describe themselves to me and tell me where they live, and what items from our inventory they want. Since you cannot build a human being object you can create a "HumanInformationInterface" object or a "HumanProxy" object that will be your interface to the real-world human. A generalization of this heuristic is a class, "DataSourceInterface," of which the "HumanProxy" is an instance. Much more will be said later about the nature of DataSourceInterface objects.

Heuristic: Look for components.

A component will look exactly like an object at this stage of development, i.e., it will have a name and a set of responsibilities. When you foreshadow those responsibilities, however, the object starts to look complicated and will seem to need a lot of collaborators, usually a sign of a need for further decomposition and redistribution of factored responsibilities.

If the set of responsibilities are strongly coherent and if domain experts seem aware of, but unconcerned by, an internal complexity - the way that people are aware that an automobile has a complex internal nature but ignore that in favor of using a car as a car - then you probably have a potential component. You should identify your object as a component and plan to to to do detailed analysis of it later, as if it were a small separate piece of application software.

[1] Software developers must also achieve these tasks. Users are suspicious and the developer needs their trust and acceptance if she is to be successful.

[2] Reflects U.S. and western culture. Other cultures use right to left and down before right or left conventions.