Chapter 3

Discovery in Principle

This chapter gives a flavour of the Discovery Method, describing its principles and how it works, compared to other approaches.

Four Principles of Discovery

The desirable properties of any method were sketched in Chapter 1.

Chapter 1 described some of the ideas that originally motivated the creation of the *Discovery Method*. Over the years, these ideas have been refined into a set of principles, which set the *Discovery Method* apart from other less guided approaches to object-oriented analysis and design. In fact, these principles are what any self-respecting analysis and design method *should* have, if it is to succeed as a *bona fide* method. There are four guiding principles, which are bound together by two overarching themes. The four principles are: *direction, selectivity, transformation* and *engagement*.

The Principle of Direction

Harnessing cognitive bias is introduced later in this Chapter.

The *Discovery Method* is a directed method, in the sense that it describes rational sequences of steps for apprentice developers to follow during analysis and design. This contrasts with unguided or round-trip approaches, which do not prescribe any particular order of development, but make general suggestions about suitable activities and otherwise rely on the discretion and skill of the designer. In the *Discovery Method*, the sense of direction is provided in several ways, by the method's harnessing of the bias in human perception, by its rule-based *discovery procedures* and by its sequencing of activities.

First examples of "discovery procedures" appear in Chapter 6.

The *Discovery Method* is so-called because every activity is codified as a *discovery procedure*, that is, a rule-based technique that is designed to uncover more information, given what is already known. Every discovery procedure has an initial trigger, a feedback loop and a completion criterion. The initial trigger may be the presence of particular information, the completion of an earlier model, or the selection of a particular architectural policy. The feedback loop is designed in such a way as to be self-reinforcing, generating more of the required information until a completion condition is met. In this way, the developer knows when the activity is finished.

Each activity may generate a diagram, a text form or actual software. These products are generically known as *deliverables*. Different activities are linked on the basis of common deliverables. The outputs of one activity serve as inputs to subsequent activities. Partial orderings of activities emerge, leading to different *discovery paths* leading from one development phase to the next. There may indeed be more than one path through certain phases, which helps to reinforce the conclusions reached when the paths converge on a common result.

The Principle of Selectivity

The eclectic generation of methods is described in Chapter xx in Part V.

The *Discovery Method* is a selective method, in the sense that it chooses its techniques and notations carefully from all those on offer, based on each technique's fitness-for-purpose and the notation's ability to direct the developer's mind towards a particular productive view of the system. This contrasts with eclectic methods, which are characterised by their uncritical borrowing of diagrams and techniques from diverse approaches, whose analytical principles may even stand in mutual conflict. In the *Discovery Method*, every technique is critically appraised, to determine its purpose and effect in the original context for which it was developed; and every diagram is ruthlessly purged of divergent notational elements until it presents a single, consistent view of the system.

Further cases of eclecticism in UML are described in an analysis of the UML notations in Chapter xx in Part V.

As an example of the kind of problem that can be caused by eclecticism, consider the UML class diagram. This subsumes a number of different modelling perspectives, drawn from different schools of design. Its attribute lists and simple associations model the same things that used to be called *entities* and *relationships* in entity-relationship modelling (ERM)[1]. On the other hand, its lists of methods and directed associations model the same kind of functional dependency graph that used to be called a *collaboration graph* in responsibility-driven design (RDD)[2]. The first of these diagrams is intended to support database design, by a process of minimising data dependency. The second of these diagrams is intended to support modular subsystem identification, by minimising functional dependencies.

Now, if developers are presented with a class diagram that gives a mixed message about the connections between the classes, how are they expected to perform the next design step? The diagram simultaneously portrays two conflicting perspectives of the system, held in unresolved tension. The developers can neither see one view, nor the other view clearly. As a result, they will be inhibited from performing a suitable design transformation. Should they minimise the data dependencies? Should they minimise the functional dependencies? Most often, they will not even be able to perceive that the system would benefit from a design transformation, since the need for it is not readily apparent in the diagram. In the end, diagrams are simply drawn and then never further analysed to improve the quality of the design, a condition sometimes known as "analysis paralysis" (*sic*).

In the *Discovery Method*, the data model is a distinct diagram from the collaboration graph. Each uses a selected subset of the UML class diagram notations, to portray a single, consistent viewpoint. Each diagram is intended to support the particular activity for which it was originally devised. So, attributes and simple associations play a part in the activity that minimises data dependency during database design, whereas collaborations (directed associations) and methods play a part in the activity that minimises functional dependency during subsystem design. To counter the weakening effects of eclecticism, the *Discovery Method* emphasises each technique's fitness-for-purpose and restricts the use of each diagram to the original focus for which it was intended.

The Principle of Transformation

The *Discovery Method* is a transforming method, in the sense that it expects the final design models to look radically different from the initial analysis models, as a result of structural and functional transformations. This contrasts with elaborating methods, which view the process of design as the gradual addition of concrete details to initial analysis models, in the spirit of the *seamless transition* hypothesis. The *Discovery Method* rejects the idea that design is simply the further elaboration of analysis models and asserts that analysis and design are separate concerns. Properly speaking, analysis is discovery, whereas design is invention; and these activities need not even be carried out within the same modelling paradigm. In the *Discovery Method*, analysis is task-oriented and design is object-oriented.

The "seamless transition" hypothesis was described in Chapter 1.

Pressing analysis models into designs results in poorly coupled systems and inflexible designs. As an example of this, consider the following fragment of a system developed for an estate agent (realtor) using the "think of an object" approach. From the real world, the developer plucks the object concepts: *House, Purchaser* and *Vendor*. These interact in the following ways: the *Vendor* puts the *House* up for sale; the *Purchaser* pays a deposit on the *House*; the *Vendor* agrees with the *Purchaser* on a sale date; the *Purchaser* pays the balance on the *House*; and the *House* transfers its ownership from the *Vendor* to the *Purchaser*. This forms a tightly knit ring of collaborations, which would require mutual references from each of these concepts to the other, in order to support the kinds of messages sent between them.

This design is poor for a number of reasons. Firstly, it is hard to establish the correct order of construction for these objects. The program would need special set-up methods to initialise the various mutually-recursive and cyclic connections in this model. Secondly, it is difficult to see where the flow of control originates. In this model, each of the object concepts appears to be responsible for initiating some of the actions. In fact, the flow of control should probably originate outside this group, in a so-far undiscovered object. Finally, the object concepts are so tied to this particular application, that they cannot be used elsewhere. The *House* object may contain references to its *Purchaser* and *Vendor*; however these are incidental and not intrinsic properties of a *House*, which properly describe such things as its location, its building style, its size, its age and its value. This means that *House* cannot be reused in other contexts that have no *Purchaser* or *Vendor*.

Techniques for generating this kind of design are described in Part III – System Modelling.

In the *Discovery Method*, the design for the same system fragment would end up looking quite different. None of *Purchaser*, *Vendor* or *House* would contain references to the other. A mediator object, called *Sale*, would link the three concepts for the duration of the transaction and the flow of control would emanate from *Sale*, which monitored the progress of the transaction. There would not exist separate *Purchaser* and *Vendor* objects, but these would be interfaces to a single *Client* object, with the advantage that the same *Client* may participate in multiple *Sales*, selling one *House* and buying a different one. The *House* concept could be reused in applications that had nothing to do with real-estate transactions, since it would model only the intrinsic properties of the building. The structure and control flow in this design may look nothing like the textual analysis of the process described two paragraphs above. But this design will be traceable back to the requirements, by a process of systematic transformations.

The Principle of Engagement

The *Discovery Method* is an engaging method, in the sense that it supports high-bandwidth communication among the stakeholders and the developers and constantly directs the focus of attention of all the participants back onto the important issues that matter. This contrasts with disengaged methods in which the only consultation occurs at the start, when the analyst meets the customer to draw up a requirements document, after which the developers produce the software independently, in isolation from the creative pressures that shaped the requirements. Engagement really covers several aspects of human interaction, including participation, communication and self-awareness.

Agile methods, such as *Extreme Programming* (XP), have recently re-emphasised the importance of continuous customer involvement[3]. This makes a lot of sense, because complex or subtle requirements take time to uncover and make explicit. As the developers produce more systematic models of the customer's requirements, they will discover logical gaps in their understanding that need clarification. Furthermore, the requirements may change, as the customer's own understanding grows of how the system's capabilities might meet the needs of the business. For this, it is essential to ensure the continuous participation of the customer. To encourage this, the *Discovery Method* actively promotes ways in which the customer can feel involved in the process of development.

A key element of this is clear communication. The *Discovery Method* promotes a continuous dialogue with the customer through early opportunities for feedback and the use of diagrams and forms to visualise and describe the evolving structure of the business model. The clarity engineered through the use of simple, user-friendly diagramming techniques fosters the desired high-bandwidth communication in the customer's own language. The customer is able to propose changes and the developers are able to propose rationalisations of the business process, in ways that both parties understand. The need for clear communication also exists within the developer team. Decisions about the direction in which a design should go are easier to make when the focus of each diagram is a single view of the system and the elements of the notation all have a clearly defined and consistent semantics.

Task Structure and Task Flow models are used in Part I – Business Modelling.

They are first introduced in Chapter 7.

Engagement means more than just clear communication, however. It also means raising self-awareness, so that the developers understand why they are doing what they do. Developers should never follow a method slavishly, but should engage constantly with it, questioning why a particular diagram or technique is useful in the current context. One of the benefits of the *Discovery Method* is that it makes apprentice developers self-consciously aware of the rationale behind the design activities they undertake, which is the first step in becoming an expert.

Two Overarching Themes

In addition to the four guiding principles, two overarching themes exert a more pervasive influence on the way in which the *Discovery Method* is practised. It seems better to refer to these as linking themes, rather than separate principles, because their effects can be seen in many of the principles. The two overarching themes are: *cognition* and *responsiveness*.

The Theme of Cognition

The *Discovery Method* tries to be sensitive to the way in which the human mind perceives and manages information. According to Gestalt theories of perception and cognition, the human mind is

The Gestalt theory of cognition was introduced in Chapter 1.

constantly trying to reduce the volume of raw information presented to it by the senses. It does this by abstracting over chunks of raw data and labelling these chunks, so that it need only process the labels. What is interesting is how these chunks are formed. The Gestalt psychologists identified some important grouping rules[4]:

- similarity – if the next fragment of data seems much like the last, assume it is part of the same larger, static phenomenon;
- common fate – if the next fragment of data forms some kind of moving pattern with the last, assume it is part of the same dynamic phenomenon;
- common onset – if a number of fragments of data seem to begin at the same moment, assume they are part of the same parallel phenomenon;

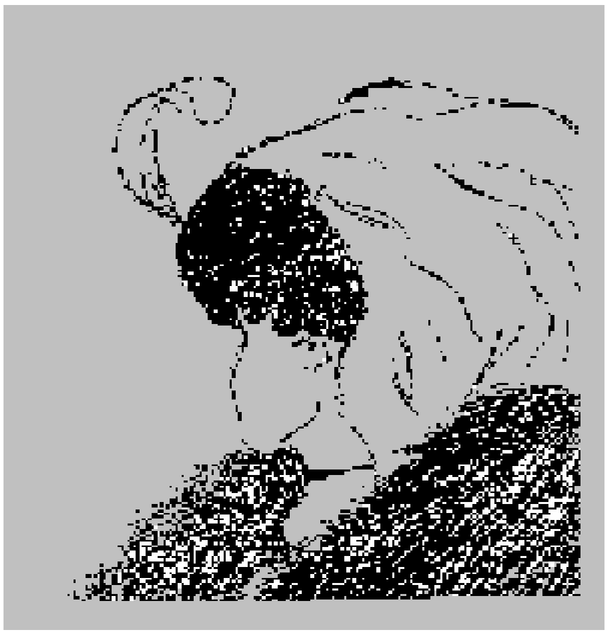

Figure 3-1: An Experiment in Perception[5]

These and other principles explain how we perceive visual images, for example. A computer image is made up of raw pixels of different colours. By the first grouping rule, we identify neighbouring pixels that have the same colour as belonging to the same coloured patch. By the second grouping rule, we identify chains of pixels that describe straight and curving lines. By the third grouping rule, we identify more complex textures, such as crosshatched shading. In the end, we do not perceive the raw pixels, but instead perceive the shapes and outlines. The “figure-ground effect”[6] describes how certain coherent shapes become detached from the background. This is the process of visual abstraction. Look at the image of a woman in figure 3-1 and see how these grouping principles apply, in the way your mind naturally and subconsciously creates patches, lines and textures out of raw pixel data.

While this grouping is absolutely necessary to simplify and interpret the raw input, it also introduces a bias, forming certain abstractions and ruling out other ones. All human perception inevitably introduces some kind of bias. This is demonstrated by the way in which we can be fooled into making categorical judgements about ambiguous raw input. This was investigated by the early Gestalt psychologists, who used ambiguous images (optical illusions) to see what their test subjects perceived in them. The image of a woman in figure 3-1 is in fact a famous ambiguous image. Stare at this again for a moment and decide how old you think this lady is, before continuing to read the following paragraphs.

You have just conducted an experiment in visual perception, with yourself as the test subject! Most people, when staring at this image, see one of two possible pictures:

- a pretty young lady, looking over her right shoulder;
- an ugly old woman, looking ahead and downwards.

Which of these did you see? The reasons why you may have seen one and not the other have to do with low-level grouping decisions. Now go back and see if you can perceive the opposite interpretation of the image. With prompting, some people can see both the old and young woman, but others cannot, since they are locked into their first perception of the image, as Gestalt theory predicts.

In both images, the woman appears to be wearing a thick fur stole and a headscarf, with a feather in her hair. However, if you focus on the area of the face, the same areas can be perceived as a chin and exposed neck in one interpretation, or as a hooked nose and angular, sunken chin in the other interpretation. The near-horizontal line in the bottom half of the image can be interpreted either as a necklace on the young lady, or as the thin pressed lips of the old woman. The rather indeterminate shape just below the hairline in the centre of the image can be interpreted as the downcast left eye of the old woman, or the left ear of the young lady.

The categorical perception of the whole image is based on early subconscious interpretations of its parts. If you thought the line was a mouth, you would tend to see the old woman. If you thought the centre shape was an ear, you would tend to see the young lady. Other parts of the image tend to reinforce one or other view. For example, the set of the shoulders tends to reinforce the sunken face of the old woman. However, the backward curl of the feather tends to reinforce the turned away face of the young lady. These influences come from the overall sense of perspective, which itself is a higher-level mental construction.

From this, we learn that our initial low-level interpretations form the building blocks for all higher-level interpretations and so bias all subsequent perception. Once our minds have established a framework, this has a filtering effect on how new data is perceived. Our minds therefore tend quite naturally to jump to early conclusions, which tend to persist. According to Gestalt theory, it is difficult to undo a first perception, because this requires more mental effort, rather like swimming upstream against the current. The human mind prefers to go with the flow, because this conserves mental resources.

The *Discovery Method* seeks to be sensitive to the natural bias and limitations of the human mind in the way ideas are formed and processed. As an example of this, the *Discovery Method* explicitly advises against creating an object model from the concepts found in the business domain during the analysis phase. This is because fixing early object abstractions is dangerous, establishing certain concepts and precluding others. The method seeks to delay the fixing of object abstractions, emphasising the plasticity of concepts until these can be properly evaluated. At other points, the method deliberately seeds certain productive concepts, rather like the seed crystal in a crystal-growing kit, around which a whole framework of objects will eventually grow. Generally, the method seeks actively to direct the focus of attention in all of its techniques.

Seeding object concepts is described in Chapter xx in Part II.

The Theme of Responsiveness

The *Discovery Method* strives to be responsive to different situations, accepting change as a natural fact of life. Other agile methods have highlighted the importance of embracing change[7]. In *Extreme Programming*, this refers to maintaining a positive attitude in the face of constantly shifting customer requirements. Rather than resist late modifications to the specification, the *XP* method accepts that such change is inevitable and responds by allowing constant modification of the code-base, supported by a rigorous retesting policy.

The ability to adapt to changing requirements is also important in the *Discovery Method*. However, this is handled in a slightly different way. One of the advantages of following a task-oriented approach in analysis is that changes in system requirements usually come in task-sized chunks, each based around a single business function. The customer may ask at a late stage to have new tasks supported by the system. This is accommodated in the architectural design, which anticipates the addition, removal and modification of major business functions (which typically access common data services). The early modularisation of business functions allows each task to be developed separately, possibly by different teams. The main delivery model is incremental, also in terms of tasks.

Elsewhere, the *Discovery Method* seeks to remove obvious barriers to change. One of the biggest, but least recognised problems in

UML-based methods is the way in which the diagrams themselves pose obstacles to change and so inhibit progress. If developers invest great effort in large diagrams with complex annotations, they will be reluctant to change these when alternative concepts are proposed. For example, the UML sequence diagram is one of the most fragile models, which has to be redrawn every time a new object is invented, or it is realised that the flow of control proceeds in a slightly different order. Large, complex and fragile kinds of diagram should be avoided. Small, simple and local diagrams should be promoted instead. It should always be possible to throw a diagram away without losing the system.

Software engineering processes are discussed in Chapter xx in Part V.

The *Discovery Method* is responsive in another way, in that it adapts to different software engineering processes and in-house styles of organisation. A single person, a developer team, or several teams working semi-independently may all benefit from following the *Discovery Method*. The method can be applied within lightweight agile processes having almost no overheads, or within more heavily monitored software engineering processes. It adapts readily to any of the waterfall, spiral or fountain software lifecycle models. The reason why it can do this is because it has no monolithic global process of its own. Rather, the method consists of sets of *discovery procedures*, which connect at common deliverables. Users of the method are free to pick different elements of the method, so long as a consistent subset of connected activities and deliverables is used.

The practices of XP are shown to be self-supporting in Chapter xx in Part V.

The *Discovery Method* is therefore more flexible than XP in the ways in which it can be adopted. As a kind of process, the main weakness of XP is that its component parts are critically interdependent. If one practice is omitted, the whole edifice collapses; and this is supported by anecdotes from programmers working in projects where XP was only partly adopted[8]. The problem is that XP rigidly prescribes the organisational process, but organisations usually want to be more flexible. On the other hand, XP is very flexible about the logical process, trusting that a suitable design will emerge, whereas this should be agreed and prescribed up-front, even if it undergoes some change during development.

Finally, the *Discovery Method* adapts to different contexts of usage and the nature of the systems engineering problem. Altogether there are more than sixty different techniques described in the *Discovery Method*, this does not mean that every project should use all of them. Some will be more appropriate and others less so, according to the needs of the project.

Review Exercises

1. In software engineering methods, why do designers produce diagrams? (Moderate)

2. What is *analysis paralysis*? (Easy)

3. How does an awareness of cognitive bias permeate the two principles of *selectivity* and *engagement*? (Hard)

4. "The Discovery Method and XP take opposing views on the organisational and design processes". Discuss. (Moderate)

5. What is the proper role of an initial requirements document in the *Discovery Method*? (Hard)

Bibliographic Notes

[1] Peter Chen devised the Entity-Relationship Modelling technique for normalising sets of data (Chen, 19xx). *Entities* grouped together sets of indivisible attributes. *Relationships* could be binary, ternary or higher and showed the multiplicity of entities at each end.

[2] Rebecca Wirfs-Brock and her colleagues devised the Responsibility-Driven Design technique for identifying and restructuring object concepts (Wirfs-Brock et al., 1990). A *collaboration* was a directed relationship between one class and another. A *collaborator* was a class that provided services to another client class.

[3] One of the famous practices of XP, this tenet used to be known as the "on-site customer". In reality, it is often hard to keep your customer on-site, as he has his own job to do (Beck, 2000).

[4] These principles were first identified by the early Gestalt psychologists investigating visual perception (Koffka, 1930). They have since been found to apply to auditory perception and general human cognition.

[5] This particular image appears in a number of different sources.

[6] The *figure-ground effect* was a term coined by the early Gestalt psychologists (Koffka, 1930) to describe our ability to detach objects from the background in visual perception.

[7] The inability of conventional methods to accept late changes in requirements is one of the driving forces behind agile methods. The call to "embrace change" is another tenet of XP (Beck, 2000).

[8] See Matt Stevens and Doug Rosenberg's detailed critique of XP in their book, *Extreme Programming Refactored: the Case Against XP* (Stevens and Rosenberg, 2002).