Chapter 4
# Discovery in a Nutshell

*This chapter gives an overview of the Discovery Method, describing the activities carried out in each of its four overlapping phases.*

## The Phases of Discovery

*Different software lifecycles are discussed in Chapter xx in Part V.*

The activities of the *Discovery Method* cover the same stages in the software lifecycle that are usually known as: *requirements capture, systems analysis, unit design, systems design, implementation, testing* and *maintenance*. However, it should not be assumed that these activities are grouped into stages that are completed in a strictly linear order. Indeed, different parts of the developing software system may progress at different rates, and new elements may always be added later to a working system prototype.

*The ordering of discovery procedures was discussed in Chapter 3.*

For this reason, we try to avoid referring to different "stages" of the *Discovery Method*, as though these were the larger steps in some overall process. Instead, the activities of the *Discovery Method* are grouped into four *phases*, which focus on different aspects of the lifecycle. The grouping relates to the purpose of the activity, not to the timing of its execution, which is controlled by different criteria.

The four phases of the Discovery Method are: *Business Modelling, Object Modelling, System Modelling* and *Software Modelling*, named according to their main emphasis[1].

## The Business Modelling Phase

*Part I covers Business Modelling.*

The *Business Modelling* phase is where a project is set in motion. It is assumed that some external customer has approached a software house, with a request to build a software system. The goal of *Business Modelling* is to explore the customer's requirements fully, capturing not merely the initial narrow specification of the system's presumed operations, but a much broader structured model of the business context in which the system will operate[2]. This allows later negotiations to determine the scope of the eventual software system, taking into account possible ways in which the customer's business could be restructured to operate more efficiently. Diagrams and text documents are produced which capture the essential business processes. The *Business Modelling* phase also generates a contract for building the system and a plan for incremental delivery, ordered by priority.

The protagonists are known as the *developer* and the *customer*. In fact, there will often be many representatives of both sides, but it is easier to think in terms of stereotypical roles. The developer and customer enter into an initial discussion to establish the grounds for wanting the software system. After this, a period of dialogue ensues between the developer and customer, in which the nature of the customer's business is explored in detail. It is most common to conduct this investigation through a series of interviews, but other kinds of interaction are possible, including holding workshops.

*Interviewing techniques are described in Chapter 6 in Part I.*

Different styles of non-directive and goal-directed interviewing are practised, triggered by signals in the discussion. It is extremely important to allow the concerns of the customer to emerge naturally and completely. It is considered very bad form for the developer to indulge in presumptive guesswork and base questions on a partial, or assumed understanding of the customer's business. The dialogue will eventually elicit:

- the critical concerns that require the software system
- what the various stakeholders want from the business
- a completely structured model of the business and workflow
- a rationale for why the business is conducted in this way

Descriptive terms use the customer's preferred language. Much of the early interviewing is spent learning this language and defining terms that describe business tasks, business stakeholders and the physical objects and documents used in the business. A dictionary of terms is constructed, using a binary patterning model that encourages tree-structured concept spaces.

*Task structure and task flow diagrams are described in Chapter xx in Part I.*

At the same time, the developer performs a task-oriented analysis of the business. In this, each major business function is modelled as a task that involves one or more of the stakeholders and affects one or more physical objects or documents used in the business[3]. The stakeholders are modelled as actors and the documents as objects. The purpose of this analysis is twofold. Firstly, it underpins one of the more important goal-directed interviewing techniques and fosters the complete exploration of the business. Secondly, it allows all relationships between tasks, actors and external objects to be established and expressed in structured models, collectively known as task diagrams.

*Narratives are described in Chapter xx in Part I.*

Alongside the task diagrams, semi-formal textual descriptions, known as narratives, are produced for each business task. Their primary function is to relate how each business task is carried out, in the preferred language of the customer, using agreed terms defined in the dictionary. Their secondary function is to capture constraints on the actors and objects involved, which affect the order in which tasks may be carried out. Sets of tasks fall naturally into partial orders, according to their pre- and postconditions. From this, different possible orderings are constructed and evaluated.

*The formal semantics of the models used in the Discovery Method are described in Chapter xx in Part VI.*

Throughout the *Business Modelling* phase, the developer presents task diagrams and narratives to the customer for comment. The customer becomes self-consciously aware of the structure and workflow of the business and prompts the developer to fill in gaps in the models. Once it is agreed that all primary information has been captured, the developer is free to propose logical rationalisations of the customer's business[4]. A process of negotiation ensues, in which non-functional concerns may play a significant part. Task diagrams are restructured, with the addition or deletion of tasks, so long as the core business is preserved. Eventually, the scope of the system is determined in the context of the revised business model. From this point, the task and narrative models constitute a formal specification for the system. They have a fully logical interpretation. They support formal reasoning and the generation of tests.

*Drawing up a contract is described in Chapter xx in Part I.*

The last issue to be decided is the schedule. Tasks are prioritised for delivery in an incremental fashion, allowing the system to be built and tested iteratively. The priority of each business task is decided using a cost-benefit matrix[5], which weights the customer's and developer's contrasting concerns. The delivery schedule is so arranged as to guarantee the completion of core components, while allowing some flexibility for optional components.

## The Object Modelling Phase

*Part II covers Object Modelling.*

The *Object Modelling* phase is where the viewpoint gradually shifts from task-oriented analysis to object-oriented design. The goal of *Object Modelling* is to identify robust candidate object concepts from the task descriptions and progressively refine these down to a collaborating society of software components. Different viewpoints are used to seed specific ideas for the front-end controller objects and the back-end data managers, which can be imposed top-down by architectural decision, if required. Further seeding for the middle tier of objects comes from catalogues of frequently used kinds of object and ideas for local collaborating clusters of objects are taken from the design patterns catalogue. The list of nouns from the dictionary of terms also provides a rich set of business metaphors that might be turned into objects. All of these are submitted to the general object modelling approach, which uses responsibility-driven design (RDD) to identify productive object concepts in a bottom-up rule governed style[6].

The input to *Object Modelling* is a set of task specifications. This set may change as the customer's requirements evolve, so it is important to establish a suitable open-ended architecture for the system, into which new tasks may be plugged at a later stage. The architecture of the system is designed around the roles of the business stakeholders who carry out the various business tasks. Initially, the interfaces used by particular actors are isolated and the tasks carried out by each actor are modularised, using the *Command* design pattern. Alternatively, if the nature of the business is highly modal, such that it is desirable to restrict the order in which tasks may be selected, then the *State* design pattern is used instead. The most commonly-occurring architecture in business information systems is one in which the top layer is characterised by command-objects and the next layer down is populated by objects that model the particular business domain, which function as gatekeepers, granting or denying permission for the top-level tasks to execute.

*User interface modelling and the top-level command structure are described in Chapter xx in Part II.*

This middle tier of objects encodes the business logic of the system. Seed object concepts come from the actors and objects named in the narratives; however, these concepts are often transformed later into more abstract components by the rules of RDD. If an actor or object is named in the preconditions for a narrative, this is strong evidence that the concept is a gatekeeper, since it can enable or prevent the execution of a task. For each such gatekeeper, a task flow diagram is found which describes the order in which this object naturally participates in the complete set of tasks that affect it. A deterministic procedure then converts the flow diagram into a state diagram for the object, by precisely inverting the nodes and arcs of the flow diagram. Gatekeeper state diagrams model important states of the business process, which enable or prevent transactions. Eventually, a coding idiom will translate these diagrams into software.

*Business logic modelling and gatekeeper objects with state are described in Chapter xx in Part II.*

The focus of attention now shifts to the data services supported by the system. Not all systems need detailed data modelling, especially if they are single-user systems, or handle only small amounts of data. For this, the object persistence mechanisms (such as Java's object serialisation) provided by the programming language may suffice. However, most business information systems rely on large amounts of data, recording the customers and suppliers of the business, together with details of contracts, sales and invoicing. All this information is usually stored in sets of data tables, where each table corresponds to a collection of objects of the same type. All of the business tasks typically access common data services, so a common interface to these is usually required.

The kinds of data to be stored are determined. Gatekeeper-objects are prime candidates for data storage, since they hold the state of the business process. Other data storage concepts come from the physical documents used in the business and also from the actors, about which the system may need to record information. There is no simple one-to-one correspondence between physical objects and software data concepts. A business document may contain multiple or repeated groups of data that would be split over several database tables, for example.

*Data models and the normalisation of databases are described in Chapter xx in Part II.*

The data model may be constructed following two quite different approaches, using either object-association modelling (OAM), or event-driven design (EDD), which also establishes a particular pattern of communication among the objects. Both approaches seek to reduce the data to a set of tables in (at least) third normal form. In OAM, associations between objects are manipulated to reduce the dependency between them and the data model is progressively normalised. In EDD, an object-event table is used to coordinate groups of objects affected by the same events, and the data model is constructed in normal form.

The output from *Object Modelling* is a fine-grained collection of candidate objects, which each perform a limited function and rely on other collaborator objects to help them fulfil their responsibilities. The reason for driving down the size of object components is twofold:

- they individually perform an obvious function;
- they are not strongly tied to their original context.

*Responsibility-driven design is described in Chapter xx in Part II.*

Responsibility-driven design is used to break down object concepts until they reach this optimum granularity. Candidate objects are proposed and named according to the roles they play in the system. Object roles are merged and split according to a detailed set of rules, until each concept manages a coherent set of responsibilities. The rules require that an object bear some responsibility for knowing, performing or enforcing in the system. There is a tendency for data objects to manage information, while control objects perform tasks and gatekeeper objects enforce constraints, although these roles overlap in certain objects. Each object should manage between two and seven responsibilities and larger objects are split up in different ways, according to a judgement about internal cohesion. The key metaphor of subcontracting is used to establish channels of communication with collaborator objects.

*Object role cards are described in Chapter xx in Part II.*

Candidate objects are logged on object role cards, which list the responsibilities, collaborators and data attributes of the concept. The advantage of doing this, over drawing some kind of communication diagram, is that the developer has a much greater freedom to insert, delete or replace objects, since the impact of each modification is relatively small. This is vital during the early stages of object identification, when concepts are plastic and the overall shape of the system is still fluid. The graph of inter-object communications can always be inferred dynamically from the current set of object role cards. Later, during the system layering activity, transformations will introduce further new objects and modify the flow of control. The object role cards also provide a valuable resource against which the functional specification expressed in the narratives can be compared. The current model of the system can be simulated by role-play, to check that all the business functionality has been captured and faithfully distributed over the set of objects. The narratives provide stories to execute and these are tested against the responsibilities owned by objects.

## The System Modelling Phase

*Part III covers System Modelling.*

The *System Modelling* phase is where the viewpoint shifts from the design of small-scale components to the optimisation of large-scale systems. The goal of *System Modelling* is twofold: to modularise the current design, identifying natural layers and subsystems; and to maintain a longer-term software investment, known as a framework. System layering techniques transform the strongly coupled graph of object roles into a weakly coupled and hierarchically layered system of classes. At the same time, the design for the current system is merged with any pre-existing software framework, which may result in modifications to the system design or to the framework. When the current design settles, the existing software framework is refactored to incorporate generic structures from the new system that have a longer-term value and useful components are also harvested.

The input to *System Modelling* is a set of object roles, which are linked by collaboration to other roles. Each object role represents an aspect of some prototypical object in one of its system interactions. It is not initially clear whether each role will eventually become a class, or an interface representing part of the behaviour of a class. The collaborations between object roles are typically dense, due to the earlier sharing out of responsibility. The next few activities manipulate the graph of object roles, now viewed as candidate classes, in order to decouple some of the more strongly connected parts of the graph.

*Collaboration diagrams are described in Chapter xx in Part III.*

One of the first activities establishes an overall picture of functional dependency between the candidate classes. This is a kind of class diagram called a collaboration diagram[7], in which each candidate class is linked by an arrow to those on which it depends for some of its behaviour. That is, an arrow is drawn from each candidate class to each collaborator that was listed on the corresponding object role card. The meaning of the arrow is a functional dependency, that is, the connection is motivated by the needs of instances of one class to invoke methods on instances of the other class. If nothing more were done, each arrow would be converted into a class reference in the code and into an object pointer at runtime.

*System layering is described in Chapter xx in Part III.*

The next activity is called system layering. It applies three kinds of design transformation[8]. With each modification, the flow of control changes and the set of object role cards is also updated. The first is the *aggregation* transformation, which seeks out strongly coupled groups of classes and closed rings in the collaboration graph. A mediator-class is invented to manage the collaboration among the closed group and connections between the members of this group are deleted. The second transformation is *server generalisation*, in which classes offering overlapping services are generalised, creating (possibly abstract) superclasses. The third and most difficult transformation is *client generalisation*, in which classes invoking similar sets of services are generalised, creating superclasses with generic algorithms. The latter two transformations merge sets of paths between classes, and so reduce the overall coupling.

*Merging with the data model is described in Chapter xx in Part III.*

During this activity, it is often found that functional dependency and data dependency exercise conflicting design forces. The optimised collaboration graph may have more connections than strictly allowed in the data model. To avoid breaking with third normal form, a novel *Query Set* design pattern is applied to support collaborations that run in a direction counter to that required by data normalisation, without increasing the overall coupling.

All of these transformations greatly reduce the number of direct collaborations, by identifying new intermediate abstractions, which eventually have a useful role to play in the system. The resulting system design is found to contain numerous instances of design patterns, such as the *Mediator, Template Method, Command, Chain of Responsibility* and *Composite* patterns, indicating the decoupled and generic quality of the design. The design has the structure of a *white-box* framework, that is, a hierarchically layered system of classes with many intermediate points that can be specialised by inheritance, to adapt it for similar kinds of application. Elements of this framework and other components are harvested for reuse.

*Frameworks are described in Chapter xx in Part III.*

A software framework typically starts to stabilise after three or more systems of the same kind have been built. Before this point is reached, the developer has to decide whether the current system should be adapted to the existing framework, or whether it is better to deliver the system according to its optimal design and schedule the immature framework for maintenance at the end of the project. Various rules and heuristics are provided to estimate the costs of different kinds of refactoring, using measures similar to those for estimating algorithmic complexity. Some examples are given of how the particular functions of a specific business may become more abstract and general operations in a longer-term framework.

A framework starts life as a white-box framework with points at which it may be specialised by subclassing, called its *hot spots*. When a framework is still evolving, this is the most productive architecture, since it allows new levels of generalisation and specialisation to emerge. Each hot spot is a partially abstract class, which expects to be specialised by overriding certain outline methods in subclasses. Such adaptation requires a detailed knowledge of the flow of control within the framework (hence the term *white-box*) and carries the risk of accidentally replacing the wrong methods. For this reason, once a framework has stabilised, it is usual to transform it into a *black-box* framework, by converting all the hot spots into strongly typed interfaces. The black-box framework is specialised more safely by inserting components with the expected methods.

## The Software Modelling Phase

*Part IV covers Software Modelling.*

The *Software Modelling* phase is where the viewpoint shifts from system-level design to actual coding and testing. The goal of the *Software Modelling* phase is to translate the optimal design faithfully into program code structures in one of the many popular object-oriented languages, such as Java, C++, Smalltalk, or Eiffel. Particular translations of some design concepts are presented as coding idioms. These may be specific to a particular programming language. The semantics of operations are enforced through assertions and testing is carried out at different levels as code modules are completed.

The choice of programming language is considered. Non-functional requirements may dictate that the system be delivered for a particular platform. Otherwise, concerns such as the availability of existing frameworks and components may play a part. Some languages provide uniform reference semantics for their objects; some provide a mixture of reference and value semantics. Some provide automatic memory-management and others require the programmer to allocate and free blocks of memory. Issues of safety and maintainability are also considered, along with the need to interoperate with other systems, such as a database or a web browser.

*Translation of references is considered in Chapter xx in Part IV.*

The classes and interfaces of the final design are translated into code outlines. While the translation of classes and attributes into code is quite straightforward, some extra consideration is given to the semantics of the inter-class references[9]. Long-term connections should remain as references, but short-term connections could be made available through method arguments. Exclusive references can be represented differently in languages with value semantics. Shared references need special treatment in languages without memory management. The order of construction for each class can be determined by examining the lifetimes of instances, to see whether these overlap, or wholly contain each other. If event-driven design (EDD) was used earlier, this information is already available. The length of object lifetimes also determines whether a class has responsibility for creating and deleting instances of other classes, or whether it simply receives them as construction arguments.

*Translation of methods is described in Chapter xx in Part IV.*

Methods are implemented for each class. Each responsibility will be refined into possibly several methods dealing with the same theme. For example, "knowing my name" may be refined into a constructor that initialises the name and an access method that allows clients to see the name. Method arguments are determined according to the earlier judgement about permanent and temporary references. Methods should be documented, preferably using a commenting style from which automatic documentation may be extracted. Third party maintainers may require complete end-to-end communication diagrams once the code has stabilised. Good documentation should describe the overall design rationale behind the software, not just document the usage of each method.

*Programming by contract is described in Chapter xx in Part IV.*

To preserve the semantics of operations, the pre- and postconditions in the narratives are converted into executable assertions. Likewise, the data invariants expressed on the object role cards are encoded. These assertions may be tested always, or conditionally. A key metaphor is *programming by contract*[10], which requires a method to deliver a valid result if it was called with valid arguments. Clear rules for checking assertions, raising and handling exceptions are suggested by this metaphor.

*Testing methods are described in Chapter xx in Part IV.*

Three kinds of testing are possible with diagrams used in the *Discovery Method*. Protocol testing is a kind of class unit testing that robustly validates sequences of method invocations, using the state diagrams as specifications. Flowgraph testing exercises all branches of major business functions, using either the narratives, or communication diagrams as specifications. Acceptance testing allows the customer to review early prototypes of the user interface, as well as the final delivered system. Tests are conducted against the narratives, as specifications.

## Review Exercises

1. Why are object role cards the best way to record object concepts in the *Discovery Method*? (Easy)

2. In an interview, the developer asks the customer, a librarian: "Tell me how you borrow and return books in your library". Why is this a bad move? (Moderate)

3. What system layering transformations might have been used to correct the ill-structured estate agent (realtor) example in Chapter 3? (Hard)

4. In a theatre seat booking system, identify a *command* object and a *gatekeeper* object. (Moderate)

5. Is identifying responsible object concepts bottom-up in conflict, or in harmony with the top-down imposition of *command* and *data* objects? (Hard)

---

## Bibliographic Notes

[1] In earlier versions of the *Discovery Method*, these were known as *Task Modelling, Object Modelling, System Modelling* and *Language Modelling* (Simons, 1998a; 1998b). The revised names hit the target concept spaces more centrally.

[2] Ivar Jacobson emphasised the importance of modelling the business context in his *Business Process Reengineering* (Jacobson, 1996).

[3] This description may sound initially similar to UML's *use cases*, but the *Discovery Method's tasks* are completely compositional, more like the tasks in SOMA (Graham, 1994).

[4] In this respect, the *Discovery Method* adopts the forgotten wisdom of older structured methods, such as SSADM (Downes et al., 198x), which spend some time on logical restructuring.

[5] This uses a spreadsheet-based technique similar to Tom Gilb's impact estimation charts (Gilb, 199x).

[6] This includes Kent Beck and Ward Cunningham's CRC-card modelling technique (Beck and Cunningham, 1989) and Rebecca Wirfs-Brock's focus on identifying objects as agents with responsibility (Wirfs-Brock and Wiener, 1989).

[7] This is the original sense in which RDD used the term *collaboration diagram*, that is, a graph linking classes to their collaborators (Wirfs-Brock et al., 1990). UML later hijacked this term to refer instead to a snapshot of a group of object instances and the messages sent between them (Booch et al., 1998). The UML2.0 specification now refers to such snapshots as *communication diagrams*, which frees up the term *collaboration diagram* again.

[8] These design transformations are adaptations of transformations originally used in the later part of RDD (Wirfs-Brock et al, 1990). Much of this early wisdom has gone unnoticed, until now.

[9] This semantic analysis of references was first suggested in the *Fusion* method (Coleman et al., 1994).

[10] Programming by contract was first realised in the *Eiffel* language by its designer, Bertrand Meyer (Meyer, 1988; 1996).