Chapter 4. An Introduction to Analysis and Design

Table of Contents

Objectives

At the end of this chapter you should be able to:

- Describe the difference between analysis and design.
- Describe how analysis and design are related.
- Describe what a software model is.
- Provide examples of models used in software analysis and design.

Introduction

The previous chapters have hopefully shown you that unless software systems are well thought out and developed using a systematic approach, they can easily become catastrophic failures. Software process models have evolved to solve this, guiding the management of software projects. This chapter discusses *system analysis* and *system design*, which are used during the generic framework activities of *communication* and *modelling* (although they may be found in the others). Analysis and design helps us to understand the problem domain the software must solve, as well as the software itself.

System analysis

The previous chapter — ??? — discussed requirements engineering, which is concerned with *what* the software is required to do. Analysis is a tool to help with this: the analyst must determine — from the problem descriptions and incomplete and informal requirements — what it is that the software should

do. This is can be an exciting and often challenging task, and focuses on describing *what* the problem is, rather than on *how* it will be solved.

In many cases, analysts build models of the existing system to help software engineers understand how the customers requesting the software are currently dealing with the problem the software should solve.

The result of the analysis is a system specification, a detailed, logical description of either the existing system, or of the new system. For the new system, this specification must satisfy the software requirements. This description must aim to fill in any gaps or ambiguities and make explicit any assumptions in the requirements.

System design

The software / system designer uses the system's specification as a starting point to determine *how* the system should achieve its requirements. Once a particular solution has been adopted, the specification is expanded and modified to clarify what must still be defined in order to be able to achieve the requirements.

The product of this activity is a complete, detailed software design.

The relation between analysis and design

Analysis and design are related activities. As such they may be thought of as two parts of the single process of converting requirements into a clear, complete and coherent software system. Thus, many of the special techniques (and computer-based tools) used during analysis carry through into design. It is for this reason that the two activities are studied together.

The following chapters of this module introduce several techniques needed for software analysis and design. These are embodied in the idea of *models*.

Introduction to models

The major artifacts of software analysis and design are *models*, a **model** being a representation of one or more aspects of the system.

The most obvious example of such a model is a map. It represents a geographical location and includes important roads, buildings and railway lines. A map does not include all detail for that location — the amount of given detail depends on the scale of a map. A map of *Great Britain*, for example, is unlikely to show every street in the country. A town map, on the other hand, deliberately shows all streets and street names, but does not show the location of all trees, plants and animals in that town.

Other examples include an architect's drawings for a building; a wiring diagram for a micro-chip; the sheet music representing a score of classical music composition. Models represent important features of the thing they model, and can be used to understand and re-create the object being modelled, the building or chip or music itself.

Models *simplify* what they model to a level of detail appropriate for its readers to understand. A road map represents enough information for a person to navigate (by car) between points on the map. The trees and plants are left out of the map because that sort of detail is inappropriate and unnecessary when navigating along streets.

This raises an important point. The model used should be appropriate to the problem. Or, rather, different models should be used for different things. A road map is good when travelling by car, but is less useful when walking in the country-side, where information about footpaths, field boundaries and contours is needed. Similarly, a micro-chip wiring diagram does not have the detail necessary to build a whole computer with disk-drives, monitors and stereo sound. A different model is needed to build a computer.

In addition to this, a person may need training before they can understand the model. Maps are reasonably clear, but even then not everyone can use them. Much specialist training is needed to understand an architect's designs; similarly, training is needed to understand the models of different aspects of a software system.

These examples serve to point the way in which we approach models for analysis and design. The main points are:

- Models are a simplified representation of a system
- Different models are used to represent different aspects of a system
- You need to learn how to understand, construct and use each particular kind of model

The method used to produce a model is called a **modelling technique**. Software which assists us to employ a modelling technique is called a **modelling tool**.

Definition of the term “model”

Two definitions of “model” from various software engineers are:

> A model is a qualitative or quantitative representation of a process or endeavour that shows the effects of those factors which are significant for the purposes being considered.
> — H. Chestnut, “Systems Engineering Tools”, Publisher: John Wiley, New York, 1965

> A model is the explicit interpretation of ones understanding of a situation, or merely of ones ideas about that situation. It can be expressed in mathematics, symbols or words, but it is essentially a description of entities and the relationships between them. It may be prescriptive or illustrative, but above all it must be useful.
> — Brian Wilson, “System: Concepts, Methodologies and Applications”, p.8, Publisher: John Wiley, New York, 1984

The properties of models

Before we can decide on what sort of models to use as an aid to software engineering, we should first determine what it is that we want the model to do for us.

The requirements for any given software system can be immensely complicated. In order to manage this complexity the system needs to be subdivided into parts, and we need to focus on the requirements for each of these parts. Having modelled a part of the system, the customer needs to confirm that the model captures what it is that they want developed. Thus the aim of this whole process is to produce a complete model of the system understandable by designers and programmers, and, where possible, the customer as well.

Many iterative and incremental process models begin development on only a portion of the software's final requirements, and perform analysis and design on this limited functionality. This software product will then be shown to the customer, who will then request changes (if any) before development on the next software increment begins. This next iteration of work will include further requirements engineering, analysis and design.

So a good modelling technique should help in the following ways:

- it considers only part of the system, focusing on particular aspects of the system
- it helps to organise the analyst's ideas
- the customer is able to understand the model, and is able to provide feedback on it
- omissions and errors should be easy to identify

- designers and programmers can produce the system from the models

Since a single model only represents part of the system, it is clear that we will need several models to help analyse and design a complete system. Given this, there are certain features which models must have in common in order to meet the above objectives.

Model properties: maintainability and disposability

In the course of developing a system, the analyst's understanding of the system will change in light of the analysis they do. The same is probably true of the customer. So in the course of analysing a system, the analyst will often make several versions of each type of model — either building upon earlier ones or even throwing old ones away and starting again. It makes sense then to have models which are reasonably cheap and can be developed and maintained without too much cost and time overhead.

For this reason the models often take the form of diagrams, pictures and text which can be printed on paper or drawn on a white-board. This makes them very cheap! Occasionally, more sophisticated models are used, such as software prototypes in the prototyping life-cycle model.

The models used should also be easy to change. Re-writing or re-drawing models by hand is tedious, and CASE software tools (Computer Aided Software Engineering tools, see ???) can be used to relieve the burden of producing models. More fundamentally though, the models themselves should be such that adding or removing detail does not have many repercussions on other, distant parts of the model. In most cases this means that the model should not represent the same piece of information in two different places. This property of reducing multiple representations of the same thing is called **minimal redundancy**. The property of changes to part of a model having minimal repercussions on other parts is called **low viscosity** or **robustness to changes**. A good model will usually have both minimal redundancy and be robust to changes.

Model properties: graphics and text

Having decided to produce models on paper, we have the choice between graphical representations of the system or textual descriptions.

It is possible to describe a system entirely in words. There are some problems with this, the biggest being that for a large system, a huge number of words will be needed. A developer trying to find technical descriptions and design decisions would be lost in a maze of words. It would also require a huge time investment for the customer to check over such a document. This is clearly not practical.

Instead, for many modelling techniques, graphics make up the largest part of the model. A diagram, by using good notation, can represent a large amount of information in a form that is easy to grasp and, more usefully, can be put on a single sheet of paper.

So what characterises good notation? Some suggestions would be that each object in the diagram means something so that there is not much unnecessary clutter caused by objects with little or no meaning. Also, the objects themselves should have well-defined features so that there is little or no ambiguity. Different types of features of a system should be represented by different symbols, so that it is immediately clear what the parts of the diagram mean.

Using graphics does not mean that no text is used at all. In some places text is the best thing to use; indeed, graphics and text can play a supporting role for each other, clarifying or explaining portions which the other would be too tedious to use (such as with use case diagrams, described in ???).

Model properties: comprehension

As discussed at the beginning of this section, the models need to be understood by many people: the customers, the developers, as well as the analysts and designers themselves. If it takes a long time to explain what the models mean, then much unnecessary time will be waisted. Good models should be clear and simple.

The ideal is for models to be well drawn and easy to understand — no explanation of them should be necessary at all. As a rule, a model drawn by an analyst which the analyst themself finds both complicated and unclear *will* be difficult to understand by others.

The main models of traditional analysis and design

Having discussed both what models are and what we expect of them, we will now introduce the models that you will be studying in the following chapters. This section is meant as an overview, so do not worry if you do not understand something. You will be introduced to each type of model in far more detail in one of the next units.

There are three key aspects which need to be considered when attempting to understand a software system:

- the data needed in the system,
- the processes or functions which the system performs
- what events occur and what changes are made over time

It is useful, therefore, to have a different modelling techniques to focus on each of these areas. Three corresponding modelling techniques are:

- Class models
- Data-flow diagrams
- Sequence diagrams

Each of these modelling techniques is a diagram-based technique with accompanying text (either on the diagram, or to supplement the diagram).

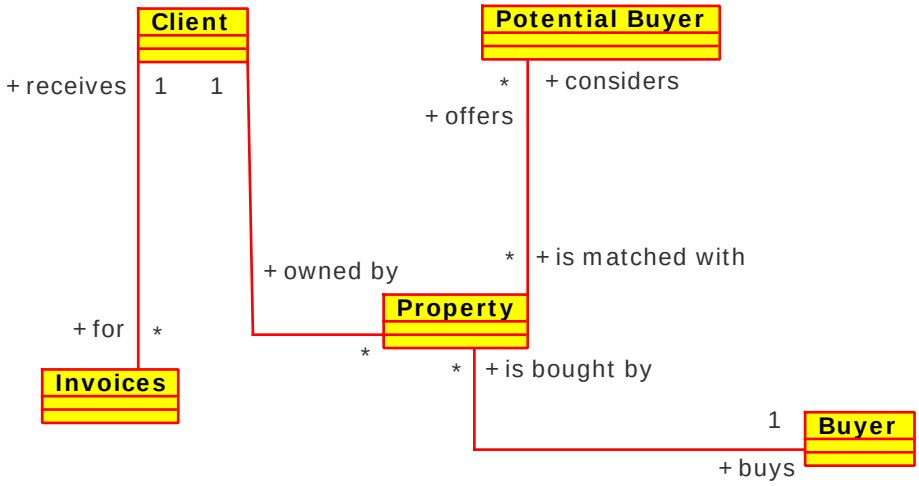
The example models presented in the following three sections have been constructed for a particular software system example. The example system is that of an estate agent that organises the buying and selling of houses. Each of the diagrams will represent a different aspect of this system.

Class model

A **class model** is used to represent the data structure of a system. In other words, the focus of this model is the data needed in the system. The data is represented as classes and relationships between classes.

An example class model is given in Figure 4.1, “An example class model of an estate agency”.

Figure 4.1. An example class model of an estate agency

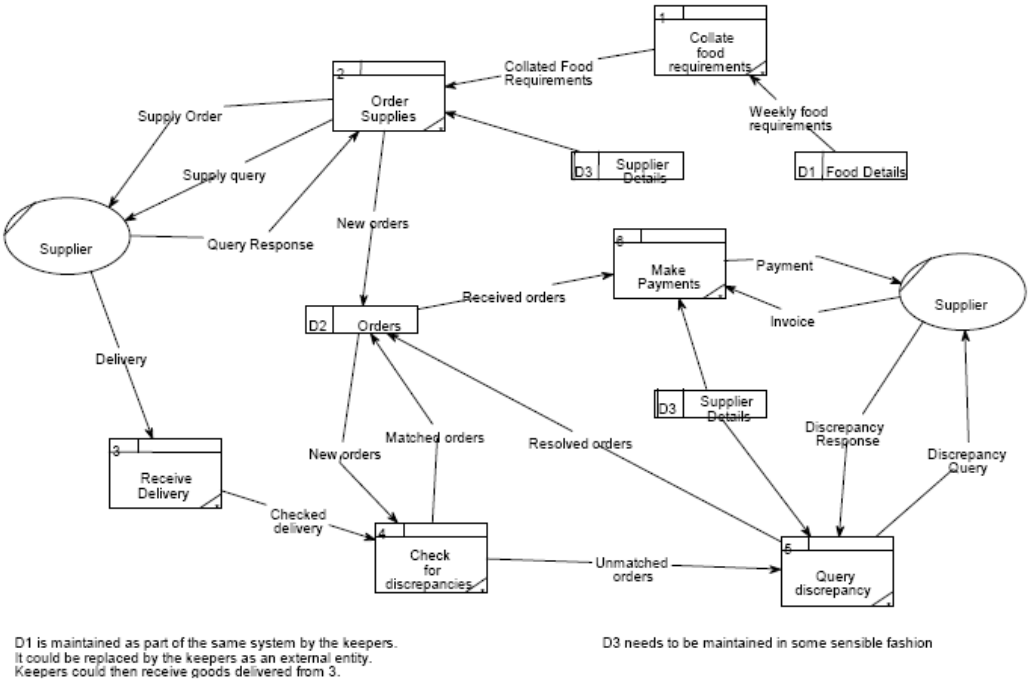


Although no details about class models have been given, it should already be possible for you to understand parts of the diagram — for example, that boxes correspond to classes such as “Client” and “Property”, and the lines represent relationships between them. Class models are a very important modelling technique, since the data and relationships of a system are least likely to change significantly. These diagrams are often constructed early on, and the other models built up from them.

Data-flow diagram

Data-flow diagrams (DFD) represent the processes or functions which the system performs, and how data flows between processes and in and out of the system. See Figure 4.2, “An example of a data-flow diagram”.

Figure 4.2. An example of a data-flow diagram



Data-flow diagrams represent a somewhat more complicated modelling technique than class models. The named boxes are processes representing actions; the named arrows represent data moving between the processes. Some boxes represent data stores where data flows either to or from. The bubbles are external entities that provide data to the system and receive the output data.

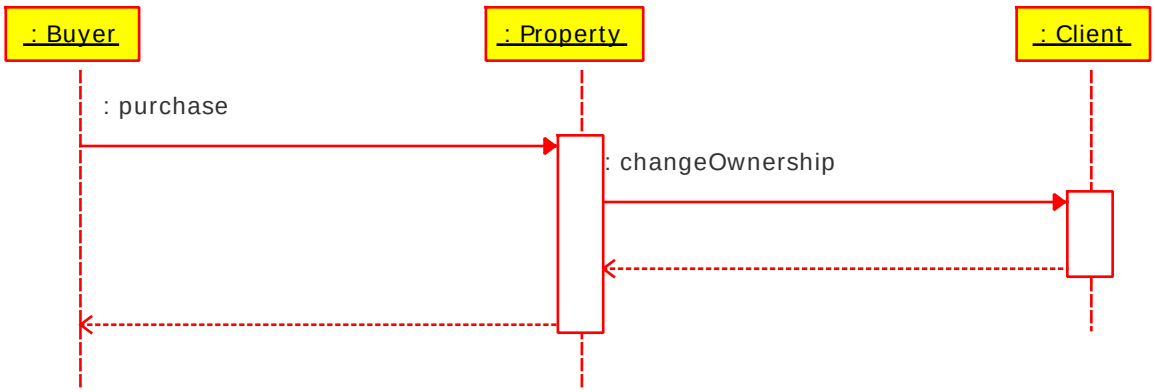
This single diagram does not demonstrate an important feature of data-flow diagrams — their hierarchical organisation. Complex processes (such as “2 Order Supplies”) can be iteratively broken into separate processes until the entire system has been described in full detail. Do not worry about the complexities of data-flow diagrams, you will learn the details of this technique in ???.

Sequence diagrams

A **sequence diagram** shows how instances of various classes (taken from, say, a class model) interact with each other over time.

The diagram should be read from top to bottom, which is the direction in which time progresses. The boxes represent not classes, but objects (which is notated using the colons in front of the class names).

Figure 4.3. An example of a sequence diagram



Useful points

All three models represent the same system, albeit different aspects of that system. This is in the same way that a drawing of a human skeleton (a “bone” model) and a model of human musculature differ from one another, but still both represent the human body.

Class diagrams, data-flow diagrams, and sequence diagrams, must relate to each other and be consistent with one another. Though they represent different things, there is a correlation between the various parts of the different diagrams. If these parts do not match up when the diagrams have been completed then it is a sign that either something has been omitted from (at least) one of the diagrams, or that they are not correct. You will learn about the relationships between the different kinds of diagrams in later chapters.

With regards to producing these diagrams for a particular software system, remember that there is no “correct” version of these models. Two different systems analysts may look at the same system and produce two different models. This is almost to be expected, since people approach the analysis of a software system from their own previous experience and, in some sense, their individual perspective gets built into the system.

Just because there is no “correct” model does not mean that all models are equally valid or equally useful. In most cases it takes many attempts to produce a “good” model. The first things written down when developing a model merely act as somewhere to start. As the model is built up the analyst's knowledge of the system increases and they may realise that some details have been omitted or that the original structure is unsuitable or unfeasible. Analysts must be prepared to insert new details, re-draw existing parts of the diagram and even, on occasion, throw away the current model and begin anew.

On modelling notation and software

Class and sequence diagrams will be drawn using the UML. The above diagrams were drawn using the Umbrello software package [http://uml.sourceforge.net], although other packages exist. You may want to explore:

- Dia [http://dia-installer.sourceforge.net]
- ArgoUML [http://argouml.tigris.org]

The SSADM notation is used for data-flow diagrams (since this is not a part of the UML). The freely available Dia package provides support for data-flow diagrams.

The benefits of using formal models

Using models in the software engineering process provides the following benefits:

- Models provides a pictorial, representation of the system, thereby providing the basis for good communication between software engineers and our customers. They are easy to draw and update, as well as being easy to check.
- Any individual model manages complexity through abstraction, by concentrating on one aspect of the system, leaving other aspects of the system to be modeled separately.
- Models impose structure on the information, providing a clear and concise representation that makes the information and its interrelationships easier to understand.
- The techniques by which models are constructed assist in highlighting areas where analysis may be incomplete.

Case studies

A case study is a short text description of an imaginary software system. It is often written in an informal manner and may contain ambiguities. As such, it resembles information that may be gathered during requirements engineering. The following “Estate Agency case study” is the case study used to produce the examples for the above diagrams.

Estate Agency case study

Clients wishing to put their property on the market visit the estate agent, who will take details of their house, flat or bungalow and enter them on a card which is filed according to the area, price range and type of property.

Potential buyers complete a similar type of card which is filed by buyer name in an A4 binder.

Weekly, the estate agent matches the potential buyer's requirements with the available properties and sends them the details of selected properties.

When a sale is completed, the buyer confirms that the contracts have been exchanged, client details are removed from the property file, and an invoice is sent to the client. The client receives the top copy of a three part set, with the other two copies being filed.

On receipt of the payment the invoice copies are stamped and archived. Invoices are checked on a monthly basis and for those accounts not settled within two months a reminder (the third copy of the invoice) is sent to the client.

Review

Questions

Review Question 1

Contrast *analysis* and *design*. How are they similar, how do they differ?

A discussion of this question can be found at the end of this chapter.

Review Question 2

What role do models play in analysis and design?

A discussion of this question can be found at the end of this chapter.

Review Question 3

Briefly outline the various properties of a model.

A discussion of this question can be found at the end of this chapter.

Answers

Discussion of Review Question 1

Analysis and design are related activities, concerned with (respectively) *what* the problem is, and *how* it will be resolved. Because of this relationship, many of the modelling techniques used in analysis are themselves used in design, and vice versa.

One large difference between them is that analysis does *not* consider any implementation details: it is merely concerned with description of existing problems. Analysis produces a specification of *the existing problem* or the *existing solution* to the problem.

Design itself can begin as a description of a *solution* to the problem, but will progress into implementation details. It also produces a specification, but this specifies the *solution* to the problem.

Discussion of Review Question 2

Models provide one of the primary means of communication of information during software engineering, especially during the activities of analysis and design. They allow us to *simplify* the information we wish to convey, and thus can *highlight* specific portions of the software system, such as the static data design (class model), user interactions (use-case diagrams) and the flow of data (data-flow diagrams). They are a tool for abstraction (see ???).

Discussion of Review Question 3

Models have the following properties:

- they consider only part of the system, focusing on particular aspects of the system
- they help to organise the analyst's ideas
- if the customer is able to understand the model, and is able to provide feedback on it
- omissions and errors should be easy to identify
- designers and programmers can produce the system from the models