Rapid Prototyping

- Prototyping software
- What is rapid prototyping?
- The rationale of prototyping
- System requirements and specification
- Strengths and weaknesses of prototyping
- Conclusions

Introduction

Before a company embarks on the manufacture of a new model of a car, it builds a prototype. The prototype is, in many respects, identical to the cars that will roll off the production line, so why bother to build it? Clearly, before starting the expensive process of setting up the production line, buying components and raw materials, and starting a marketing campaign, the company needs to make sure that the car will sell, will be easy to manufacture, and be mechanically reliable.

The risks of moving from a set of engineering drawings to full production, without any intervening stages, are high. If, for example, the new gearbox design is wrong, then the error will have to corrected, and all the models sold recalled. Such problems are not unknown in the car industry. However, the risks of gearbox failure can be considerably reduced, at the cost of building a pre-production prototype which can be tested.

Likewise, the risk of the car not selling is also a problem. Maybe the body shape or the interior design are unattractive to potential car buyers. These factors, too, need to be tested before the expensive process of production can begin. The prototype can be used by the marketing teams to gauge customer reaction.
So, when developing a new model of car, the prototype is used to reduce the risk of failure. Why then is it unusual to produce a prototype when developing a software system?

Rapid Prototyping

The cost of producing a car is largely tied up with the manufacturing process. However, the cost of software manufacture is negligible in comparison with the cost of design and development. This is the key difference — to build a prototype car, almost identical with the planned model, is the cheap part of the whole process; to build a prototype software system, almost identical with the planned system, is seen as the expensive part of the whole process. The natural argument follows: why not simply reproduce the prototype, and sell it? Why is the prototype not just like the real thing?

The trap of equating the prototype to the real thing is one that must be avoided Indeed, when introducing prototyping into a company, the arguments against this must be well-practised. There are many ways in which the prototype can be different from the planned system, and each can used to reduce the risks involved with the production of a software system.

The purpose of this chapter

This chapter defines rapid prototyping in software. Its purpose is to make the reader aware of what prototyping is, why it works, what parts of software engineering it can help with, and its expected benefits and weaknesses.

General organisation

The first section of this chapter defines rapid prototyping as a model of the planned system. It explains how this model differs from the final product, where prototyping can fit into the traditional lifecycle model, and what some of the key aspects of prototyping are.

Section 2.2 examines the rationale of prototyping, ie Why it works. In particular it relates prototyping to problem solving techniques.

In section 2.3 the problems of system requirements and specifications are discussed. This is seen as one of the hardest parts of the software lifecycle to get right, and the one in which prototyping can help the most.

The strengths and weaknesses of prototyping are discussed in section 2.4. are particularly important when it comes to deciding whether to adopt a prototyping strategy.

2.1 What is rapid prototyping?

Software prototyping is, as already discussed, one of many promising ideas in software development. This section introduces a formal definition of prototyping, and looks at the various types of prototype, and how they can be used

A definition of a prototype

A prototype is a software model of system behaviour which can be used to understand the proposed system, or certain aspects of it, and to clarify the requirements.

The use of the word 'model' needs explaining. A prototype is a representation of a system, something that is not the complete system, but has the important characteristics of the final System. The expectations of what these characteristics are, and how the prototype will be used, varies according to:

- the user of the prototype
- the information that is needed
- the application area

For example, in a well-defined application, the requirements may be known, but there is still a need to prototype the user interface, as the customer is not quite certain of the 'look and feel' of the proposed system. This type of prototype can have dummy functions, and may be thrown away once the user interface has been defined. In a much more complex system, the prototype can be used to gain information on critical system components or aspects of the design. For example, the database could be prototyped to get a feel for the design, and an estimate of expected access times. Any serious problems can be identified before they can cause trouble.

Although the prototype may just be a model, and be thrown out later, a system prototype can be the first step in the implementation process. Successive iterations will add more detail and functionality, and the early are available for evaluation, the refinements can be guided by the feedback from demonstrations to the customer. The customer may find that the early prototypes are useful in themselves, and the completed system will not give the customer any unpleasant surprises.

How rapid is rapid prototyping?

The commonplace use of the term 'rapid' in conjunction with prototyping implies that the prototype should be completed as quickly as possible. However, the value of a prototype is in the degree to which it helps meet the system goals, and not in the speed of development.

As a fundamental reason for prototyping is to understand the system goals, this must be done before the development can proceed, need for this information determines how rapid the prototype should be. In some cases the speed of writing the prototype is traded for the quality of the prototype, and the prototype is ultimately thrown out. In other cases the prototype is used to investigate a particular problem, or to elicit information, and the prototyping continues as long as the problem remains unresolved.

2.1.1 The prototype as a model

A prototype can be regarded as a model of some system that one might hope to build in the future; it is a cut-down version of the real system. A scale of a bridge has many of the characteristics of the actual bridge, with one notable exception — that of its physical size. In the same way, a software prototype may be regarded as a scale model of some other system, in that it has many of the same characteristics of that system — with some notable exceptions. The model of bridge may be used to determine some of the characteristics of the real bridge — the thickness of girders, or how the components fit together. Similarly, the software prototype may be used to determine some characteristics of the real system, the structure Of the database, or the commands available through the user interface. The model of the bridge may be used to infer information about the real bridge, such as its strength or stability. The software prototype may also be used to infer information the real system, such as the number of code modules, or the size of the data structures.

So in the same way that a physical model has some of the characteristics of the real thing and can used to estimate some of the others, the software model, or prototype, also has some of the characteristics of the real system, and can be used to estimate others.

A scale model of a bridge is noticeably different from the real thing in its size. Clearly, there is no direct equivalent to a change In physical size in software, so there the analogy ends.

There are many characteristics that can be chosen as the key differences between a prototype and the system it is attempting to model:

- user functionality
- underlying complexity
- user interface
- simpler data structures
- performance
- robustness
- hardware
- toolsets

User functionality

By reducing the functionality available to the user it is possible to include the essential functions of the system in the prototype. This allows the most important parts of the system to be tested without going to the expense of building a complete system. If a system that is used frequently is analysed, it will probably be found that about 20 per cent of the functions available are used about 80 per cent of the time. The other 20 per cent of the time is spent on the remaining 80 per cent of the functions. This '80-20' ruler is not always applicable, but it is surprising how often it turns out to be approximately true. If it is possible to predict the major 20 per cent of functions and build those into a prototype, then the goal of a rapid and cheap prototype is achieved.

The 80-20 rule is an approximation of the law first observed by Zipf (1949), who noticed that in natural language text, the nth most frequently occurring word seems to occur with a frequency proportional to l/n. The 80-20 rule is also observed in file systems, where 80 percent of transactions deal with the most active 20 per cent of the records in the file.

Underlying complexity

The underlying functionality of the system can be reduced, as opposed to reducing the functionality available to the user. For example, a multi-user system can be built in single-user mode only — thus avoiding the need to worry about the problems of data locking and other cases of multiple access to the same resource. A second example is that of a system that is designed to be distributed. A prototype can be built which resides on a single computer system, thus avoiding the problems of the communications and their consequences.

User interface

A prototype that only has a user interface is important in showing potential users how the system might look. Screen layouts can be empirically tested, command names derived, usage of colour determined, etc.

These prototypes have little functionality under the facade of the user interface. The system commands can be entered, and example output can be displayed, but little or no processing goes on underneath. The simplest example of this type of prototype is a paper-based storyboard. Here the user interface is drawn onto sheets of paper and shown sheet by sheet.

Simpler data structures

Some applications rely on complex data structures to achieve their tasks. A lot of effort can be put into making these structures efficient. A common example of this is in a relational database. During the designing of a system, the data structures in a relational database have to be put through a series of transformations known as normalisation.

This ensures the integrity of the database later on. If this stage is missed out during the building of a prototype, the database may work sufficiently well to demonstrate the principles of the proposed system, although the data cannot always be relied on.

Performance

A prototype can be a system with low performance. Whereas the final system reacts fast by using the most efficient data structures and algorithms, the prototype can be a slow system because one of the main aims is to build it quickly and cheaply. It is not necessary for a system to be fast to see if the user interface looks right, or to see if the database can hold all the relevant data. A prototype can also be used to explore performance aspects of designs, particularly in non-deterministic systems.

Robustness

Anyone who has written a significant piece of user interface code will know that a large amount of it is devoted to checking the validity of the user input. The 80-20 rule also applies here; 20 per cent of the code provides the functionality, the other 80 per cent provides the error checking (Mills, 1988). The same sort of checking goes on between any two items of software that are developed by different people. For example, a library module will generally perform a significant amount of checking on the parameters it is given. A prototype built without these checks can be built at far less cost than the real system.

Hardware

A prototype could be built on a different hardware base. The final product may be intended for a mainframe, but a prototype could be built on a personal computer (PC). This could be an advantage if, for instance, a good software tool was available on the PC with which the prototype might be built. The aim is to build rapidly and cheaply, and a special tool in a PC environment might be very useful. The PC system could then be demonstrated to the customer,

Toolsets

In order to build a prototype quickly, it is sometimes helpful to use a toolset that one might not intend to use in the final product. The tools for building the prototype may have some detrimental effect that would render the prototype useless for the real product. For example, some languages can be used to develop applications very quickly, but may produce inefficient code. The tools might be too expensive to sell with each copy of the product. This does not apply to compiled languages, but only to tools forming part of the runtime system.

Summary

A prototype may exhibit any or all of these characteristics. For example, a prototype might be built that had a complete user interface, together with a full implementation of the major system functions. The minor functions would not be implemented. Furthermore, the major functions could be built without proper error checking of the users' input. This prototype might be used to demonstrate to customers or users in order to obtain feedback.

The concept that people have of a prototype may not be correct. It is important that this is recognised and understood by those involved. It is no good creating a prototype on hardware X, and not explaining to a potential user that the system is actually being released on hardware Y. If one did, the knowledge gained from the exercise may be somewhat dubious, although this does depend on the use of the prototype.

2.12 Using the prototype

The prototype must serve a purpose. The important thing about using a prototype is that it serves to gain some knowledge. It is used for getting ideas, clarifying ideas, and confirming thoughts. Unless some knowledge can be gained from it, there would have been no purpose in building it in the first place; it would have been more valuable to develop the final system straightaway, **In order to gain knowledge, some experiments must be performed and the results monitored.** There are various ways in which the prototype can be experimented with:

- Showing it to potential users and recording their reactions and comments.
- Releasing it to potential users for a trial period, recording their comments, and recording the functions they use.
- Releasing it as a working system with reduced functionality, and similarly recording comments and functions used.
- Developers can run it and measure some characteristics, such as performance.

The knowledge gained from such experiments can be useful for developers in order to assess the specification or design, for managers in order to see if the work is progressing according to the plans, and for users and customers to see if this is what they really want.

2.1.3 Prototyping and the lifecycle model

In order to understand how prototyping can be applied to the problems of software development, it is necessary to return to the software lifecycle model. Figure 2.1 shows this in a slightly different perspective to that discussed earlier.

The lifecycle starts in step 1 with a definition of the problem to be solved. Once there is a sufficient understanding of the problem, a requirements specification can be written.

This is shown in step 2 as an empty box, since none of the internal system details are known at this point.

Step 3 specifies the functionality of the system, represented in the diagram by input and output arrows.

Again, the internal system details are not defined. When the functional specification is complete, the design process can start. The design specifies how the system should work, down to the level of the internal system components.

These are shown in step 4 by the internal boxes.

Details of these components are resolved in step 5 during the implementation of the design.

Finally, in step 6, the completed system is delivered to the customer.

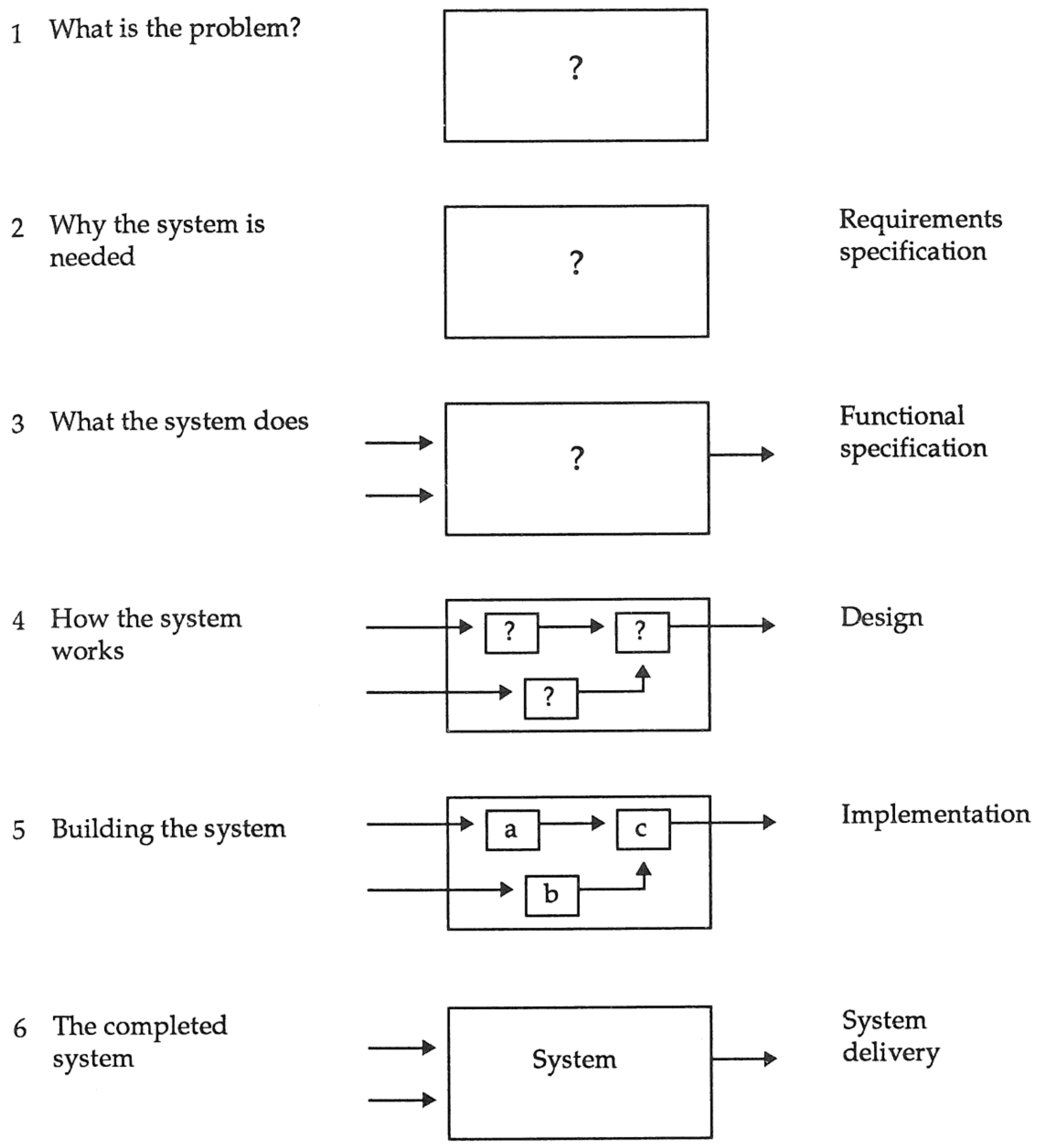

Figure 2.1 Conventional software lifecycle, illustrating the processes involved at each stage