8 Software prototyping

Objectives

The aims of this chapter are to explain how software prototyping is used in the software process and to describe different approaches to prototype development. When you have read this chapter, you will:

- understand the role of prototyping in different types of development project;
- understand the difference between evolutionary and throw-away prototyping;
- have been introduced to three different techniques of prototype development, namely very high-level language development, database programming and application and component reuse;
- understand why prototyping is the only viable technique for user interface design and development.

Contents

Software customers and end-users usually find it very difficult to express their real requirements. It is almost impossible to predict how a system will affect working practices, how it will interact with other systems and what user operations should be automated. Careful requirements analysis along with systematic reviews of the requirements help to reduce the uncertainty about what the system should do. However, there is no real substitute for trying out a requirement before agreeing to it. This is possible if a system prototype is available.

A prototype is an initial version of a software system which is used to demonstrate concepts, try out design options and, generally, to find out more about the problem and its possible solutions. Rapid development of the prototype is essential so that costs are controlled and users can experiment with the prototype early in the software process.

A software prototype supports two requirements engineering process activities:

1. *Requirements elicitation* System prototypes allow users to experiment to see how the system supports their work. They get new ideas for requirements and can find areas of strength and weakness in the software. They may then propose new system requirements.

2. *Requirements validation* The prototype may reveal errors and omissions in the requirements which have been proposed. A function described in a specification may seem useful and well defined. However, when that function is used with others, users often find that their initial view was incorrect or incomplete. The system specification may then be modified to reflect their changed understanding of the requirements.

Prototyping can be used as a risk analysis and reduction technique (see Chapter 4). A significant risk in software development is requirements errors and omissions. The costs of fixing requirements errors at later stages in the process can be very high. Experiments have shown (Boehm *et al.*, 1984) that prototyping reduces the number of problems with the requirements specification. Furthermore, the overall development costs may be lower if a prototype is developed.

Prototyping is therefore part of the requirements engineering process. However, the distinction between prototyping as a separate activity and mainstream software development has blurred over the past few years. Many systems are now developed using an evolutionary approach where an initial version is created quickly and modified to produce a final system. An iterative process model such as incremental development (discussed in Chapter 3) may be used in conjunction with a language designed for rapid application development. Therefore, the techniques used for developing a prototype for requirements validation may also be used for developing the software system itself. I discuss this in section 8.1.

As well as allowing users to improve the requirements specification, developing a system prototype may have other benefits:

1. Misunderstandings between software developers and users may be identified as the system functions are demonstrated.

2. Software development staff may find incomplete and/or inconsistent requirements as the prototype is developed.
3. A working, albeit limited, system is available quickly to demonstrate the feasibility and usefulness of the application to management.
4. The prototype may be used as a basis for writing the specification for a production-quality system.

Developing a prototype usually leads to improvements in the specification of the system. Once a prototype is available, it can also be used for other purposes (Ince and Hekmatpour, 1987):

1. *User training* A prototype system can be used for training users before the final system has been delivered.
2. *System testing* Prototypes can run 'back-to-back' tests. The same test cases are submitted to the prototype and to the system under test. If both systems give the same result, the test case has not detected a fault. If the results differ, it may mean that there is a system fault and the reasons for the difference should be investigated.

In a study of 39 different prototyping projects, Gordon and Bieman (1995) found that the benefits of using prototyping in the software process were:

1. improved system usability;
2. a closer match of the system to the user needs;
3. improved design quality;
4. improved maintainability;
5. reduced development effort.

Their study therefore suggests that the improvements in usability and better user requirements which stem from using a prototype do not necessarily mean an overall increase in system development costs. Prototyping usually increases costs in the early stages of the software process but reduces later costs. The main reason for this is that rework during development is avoided as customers request fewer system changes. However, they found that a negative consequence of prototyping was that overall system performance is sometimes degraded as inefficient prototype code is reused.

A process model for prototype development is shown in Figure 8.1. The objectives of prototyping should be made explicit from the start of the process. These may be to develop a system to prototype the user interface, to develop a system to validate functional system requirements or to develop a system to demonstrate the feasibility of the application to management. The same prototype cannot meet all objectives. If objectives are left implicit, management or end-users may misunderstand the function of the prototype. Consequently, they may not get the benefits that they expected from the prototype development.

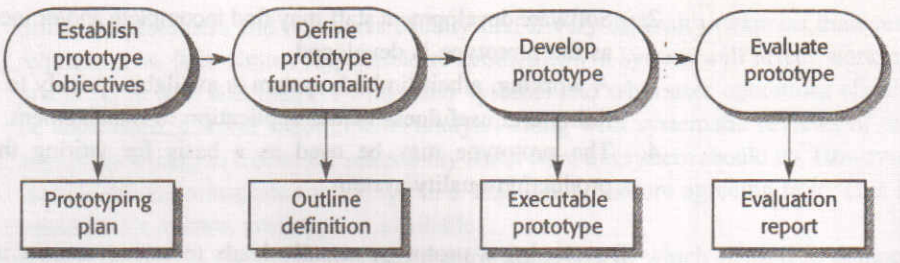

Figure 8.1 The process of prototype development

The next stage in the process to is decide what to put into and, perhaps more importantly, what to leave out of the prototype system. To reduce prototyping costs and accelerate the delivery schedule, you may leave some functionality out of the prototype. You may decide to relax non-functional requirements such as response time and memory utilisation. Error handling and management may be ignored or may be rudimentary unless the objective of the prototype is to establish a user interface. Standards of reliability and program quality may be reduced.

The final stage of the process is prototype evaluation. Ince and Hekmatpour suggest that this is the most important stage of prototyping. Provision must be made during this stage for user training and the prototype objectives should be used to derive a plan for evaluation. Users need time to become comfortable with a new system and to settle into a normal pattern of usage. Once they are using the system normally, they then discover requirements errors and omissions.

8.1 Prototyping in the software process

As I have already discussed, it is difficult for end-users to anticipate how they will use new software systems to support their everyday work. If these systems are large and complex, it is probably impossible to make this assessment before the system is built and put into use.

One way of tackling this difficulty is to use an evolutionary approach to systems development. This means giving the user a system which is incomplete and then modifying and augmenting it as the user requirements become clear. Alternatively, a deliberate decision might be made to build a 'throw-away' prototype to help requirements analysis and validation. After evaluation, the prototype is discarded and a production-quality system built. Figure 8.2 illustrates both of these approaches to prototype development.

Evolutionary prototyping starts with a relatively simple system which implements the most important user requirements. This is augmented and changed as new requirements are discovered. Ultimately, it becomes the system which is required. There is no detailed system specification and, in many cases, there may not be a formal

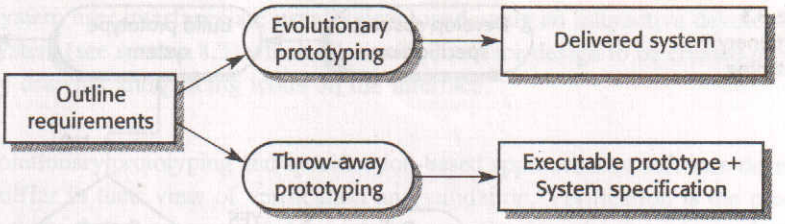

Figure 8.2 Evolutionary and throw-away prototyping

requirements document. Evolutionary prototyping is now the normal technique used for web-site development and e-commerce applications.

By contrast, the throw-away prototyping approach is intended to help refine and clarify the system specification. The prototype is written, evaluated and modified. The prototype evaluation informs the development of the detailed system specification which is included in the system requirements document. Once the specification has been written the prototype is no longer useful and is thrown away.

There is an important difference between the objectives of evolutionary and throw-away programming:

1. The objective of evolutionary prototyping is to deliver a working system to end-users. This means that you should normally start with the user requirements which are best understood and which have the highest priority. Lower priority and vaguer requirements are implemented when and if they are demanded by the users.
2. The objective of throw-away prototyping is to validate or derive the system requirements. You should start with those requirements that are not well understood because you need to find out more about them. Requirements that are straightforward may never need to be prototyped.

Another important distinction between these approaches is in the management of the quality of the systems. Throw-away prototypes have, by definition, a very short lifetime. It must be possible to change them rapidly during development but long-term maintainability is not required. Poor performance and reliability may be acceptable in a throw-away prototype so long as it fulfils its principal function of helping with the understanding of requirements.

By contrast, prototypes which evolve into the final system should be developed to the same organisational quality standards as any other software. They should have a robust structure so that they are maintainable for many years. They should be reliable and efficient and they should conform to relevant organisational standards.

8.1.1 Evolutionary prototyping

Evolutionary prototyping is based on the idea of developing an initial implementation, exposing this to user comment and refining this through many stages until an adequate system has been developed (Figure 8.3). This approach to development

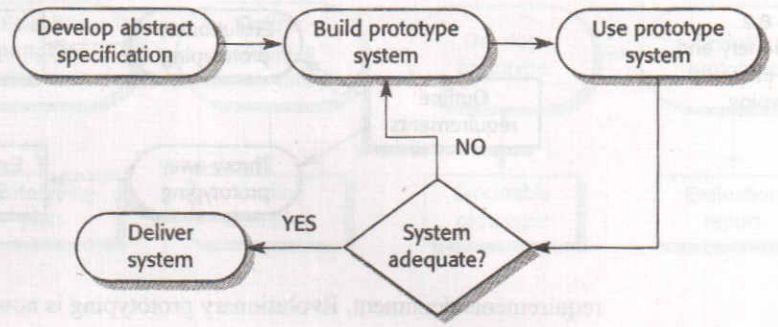

Figure 8.3 Evolutionary prototyping

was used initially for those systems (such as AI systems) which are difficult or impossible to specify. However, it has now become a mainstream technique of software development. Evolutionary prototyping is part of or has much in common with techniques of rapid application development (RAD) and Joint Application Development (JAD) (Millington and Stapleton, 1995; Wood and Silver, 1995; Stapleton, 1997).

There are two main advantages to adopting this approach to software development:

1. *Accelerated delivery of the system* As I discussed in the introduction to the book, the pace of business change means that it is essential that software support is made available quickly. In some cases, rapid delivery and usability is more important than details of functionality or long-term software maintainability.
2. *User engagement with the system* The involvement of users with the development process does not just mean that the system is more likely to meet their requirements. It also means that the end-users of the system have made a commitment to it and are likely to want to make it work.

There are differences in detail between the particular methods of rapid software development but they all share some fundamental characteristics:

1. The processes of specification, design and implementation are interleaved. There is no detailed system specification and the design documentation produced usually depends on the tools used to implement the system. The user requirements document only defines the most important characteristics of the system.
2. The system is developed in a series of increments. End-users and other system stakeholders are involved in designing and evaluating each increment. They may propose changes to the software and new requirements which should be implemented in a later version of the system.
3. Techniques for rapid system development are used (see section 8.2). These may include CASE tools and fourth-generation languages.

4. System user interfaces are usually developed using an interactive development system (see section 8.3) which allows the interface design to be created quickly by drawing and placing icons on the interface.

Evolutionary prototyping and specification-based approaches to software development differ in their view of verification and validation. Verification is the process of checking that a program conforms to its specification. As there is no detailed specification for the prototype, verification is therefore impossible.

Validation should demonstrate that the program is suitable for its intended purpose rather than its conformance to a specification. This is also difficult without a detailed specification as there is no explicit statement of purpose. The end-users involved in the process may be happy with the system, but other stakeholders may feel excluded and unsatisfied that the system meets their purposes.

Verification and validation of a system which has been developed using evolutionary prototyping can only therefore check if the system is adequate, that is, if it is good enough for its intended purpose. Adequacy, of course, is not readily measurable and only subjective judgements of a program's adequacy can be made. This does not invalidate its usefulness; human performance cannot be guaranteed to be correct but we are satisfied if performance is adequate for the task in hand. However, as I discuss below, this does cause problems where systems are developed for customers by an external software development contractor.

There are three main problems with evolutionary prototyping which are particularly important when large, long-lifetime systems are to be developed:

1. *Management problems* Software management structures for large systems are set up to deal with a software process model that generates regular deliverables to assess progress. Prototypes evolve so quickly that it is not cost-effective to produce a great deal of system documentation. Furthermore, rapid prototype development may require unfamiliar technologies to be used. Managers may find it difficult to use existing staff because they lack these skills.

2. *Maintenance problems* Continual change tends to corrupt the structure of the prototype system. This means that anyone apart from the original developers is likely to find it difficult to understand. Furthermore, if specialised technology is used to support rapid prototype development this may become obsolete. Therefore, finding people who have the required knowledge to maintain the system may be difficult.

3. *Contractual problems* The normal contractual model between a customer and a software developer is based around a system specification. When there is no such specification, it may be difficult to design a contract for the system development. Customers may be unhappy with a contract which simply pays developers for the time spent on the project as this can lead to function creep and budget overruns; developers are unlikely to accept a fixed-price contract as they cannot control the changes requested by the end-users.

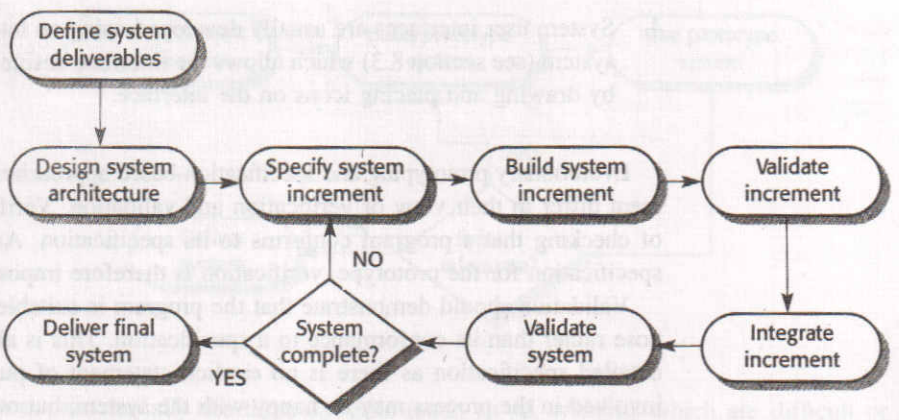

Figure 8.4
An incremental development process

These difficulties mean that customers must be realistic about the use of evolutionary prototyping as a development technique. It allows small and medium-sized systems to be developed and delivered rapidly. System development costs may be reduced and usability is improved. If users are involved in the development, it is likely to be appropriate for their real needs. However, organisations who use this approach must accept that the lifetime of the system will be relatively short. As maintenance problems increase, the system will have to be replaced or completely rewritten. For large systems which may involve a number of different subcontractors, the management problems of evolutionary prototyping become intractable. Where prototypes are developed for parts of these systems, they should be throw-away prototypes.

Incremental development (Figure 8.4) avoids some of the problems of constant change which characterise evolutionary prototyping. An overall system architecture is established early in the process to act as a framework. System components are incrementally developed and delivered within this framework. Once these have been validated and delivered, neither the framework nor the components are changed unless errors are discovered. User feedback from delivered components, however, can influence the design of components scheduled for later delivery.

Incremental development is more manageable than evolutionary prototyping as the normal software process standards are followed. Plans and documentation must be produced for each system increment. It allows some user feedback early in the process and limits system errors as the development team are not concerned with interactions between quite different parts of the software system. Once an increment has been delivered, its interfaces are frozen. Later increments must adapt to these interfaces and can be tested against them.

8.1.2 Throw-away prototyping

A software process model based on an initial throw-away prototyping stage is illustrated in Figure 8.5. This approach extends the requirements analysis process with

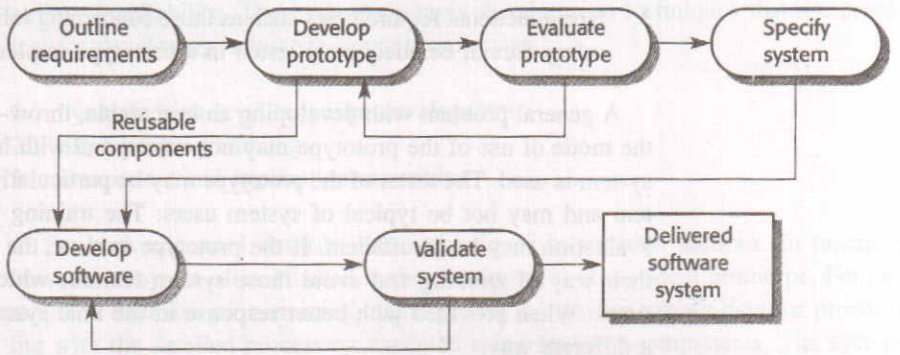

Figure 8.5
A software process with throw-away prototyping

the intention of reducing overall life-cycle costs. The principal function of the prototype is to clarify requirements and provide additional information for managers to assess process risks. After evaluation, the prototype is thrown away. It is not used as a basis for further system development.

This approach to system prototyping is commonly used for hardware systems. The prototype is used to check the design before expensive commitments to manufacturing the system have been made. An electronic system prototype may be developed using off-the-shelf components before investment is made in special-purpose integrated circuits to implement the production version of the system.

However, a throw-away software prototype is not normally used for design validation but to help develop the system requirements. The prototype design is often quite different from that of the final system. The system must be developed as quickly as possible so that users can feed back their prototype experience to the development of the system specification. Functionality may be stripped from the throw-away prototype where these functions are well understood, quality standards may be relaxed and performance criteria ignored. The prototype development language will often be different from the final system implementation language.

The process model in Figure 8.5 assumes that the prototype is developed from an outline system specification, delivered for experiment and modified until the client is satisfied with its functionality. At this stage, a phased software process model is entered, a specification is derived from the prototype and the system reimplemented in a final production version. Components from the prototype may be reused in the production-quality system so development costs may be reduced.

Rather than derive a specification from the prototype, it is sometimes suggested that the system specification should be the prototype implementation itself. The instruction to the software contractor should simply be 'write a system like this one'. There are several problems with this approach:

1. Important features may have been left out of the prototype to simplify rapid implementation. In fact, it may not be possible to prototype some of the most important parts of the system such as safety-critical functions.
2. An implementation has no legal standing as a contract between customer and contractor.

3. Non-functional requirements such as those concerning reliability, robustness and safety cannot be adequately tested in a prototype implementation.

A general problem with developing an executable, throw-away prototype is that the mode of use of the prototype may not correspond with how the final delivered system is used. The tester of the prototype may be particularly interested in the system and may not be typical of system users. The training time during prototype evaluation may be insufficient. If the prototype is slow, the evaluators may adjust their way of working and avoid those system features which have slow response times. When provided with better response in the final system, they may use it in a different way.

Developers are sometimes pressurised by managers to deliver throw-away prototypes for use, particularly when there are delays in delivering the final version of the software. However, this is usually unwise for the following reasons:

1. It may be impossible to tune the prototype to meet non-functional requirements such as performance, security, robustness and reliability requirements which were ignored during prototype development.
2. Rapid change during development inevitably means that the prototype is *undocumented*. The only design specification is the prototype code. This is not good enough for long-term maintenance.
3. The changes made during prototype development will probably have degraded the system structure. The system will be difficult and expensive to maintain.
4. Organisational quality standards are normally relaxed for prototype development.

Throw-away prototypes do not have to be executable software prototypes to be useful in the requirements engineering process. Paper-based mock-ups of the system user interface (Rettig, 1994) have been shown to be effective in helping users refine an interface design and work through usage scenarios. These are very cheap to develop and can be constructed in a few days. An extension of this technique is a 'Wizard of Oz' prototype where only the user interface is developed. Users interact with this interface but their requests are passed to a person who interprets them and outputs the appropriate response. These approaches to prototyping are discussed in Sommerville and Sawyer (1997).

8.2 Rapid prototyping techniques

Rapid prototyping techniques are development techniques which emphasise speed of delivery rather than other system characteristics such as performance, maintain-

ability or reliability. There are three rapid development techniques that are practical for developing industrial-strength prototypes:

1. dynamic high-level language development;
2. database programming;
3. component and application assembly.

For convenience, I describe these techniques in separate sections. In practice, however, they are often all used in the development of a system prototype. For example, a database programming language may be used to extract data for processing with the detailed processing executed using reusable components. The system user interface may be defined using visual programming. Luqi (1992) describes how such a mixed approach was used to create a prototype of a command and control system.

Prototype development is now usually supported through a set of tools which includes support for at least two of these techniques. For example, the Smalltalk VisualWorks system supports a very high-level language and provides many reusable components which may be included in applications. Lotus Notes includes support for database programming using a very high-level language and reusable components which may be linked to database operations.

Most prototyping systems now support a visual programming approach where some or all of the prototype is developed interactively. Rather than write a sequential program, the prototype developer manipulates graphical icons representing functions, data or user interface components and associates processing scripts with these icons. An executable program is generated automatically from the visual representation of the system. This simplifies program development and reduces prototyping costs. I discuss visual programming in more detail in section 8.2.3 which covers component and application reuse.

8.2.1 Dynamic high-level language development

Dynamic high-level languages are programming languages which include powerful run-time data management facilities. These simplify program development because they reduce many problems of storage allocation and management. The language system includes facilities which normally have to be built from more primitive constructs in languages like Ada or C. Examples of very high-level languages are Lisp (based on list structures), Prolog (based on logic) and Smalltalk (based on objects).

Until relatively recently, very high-level dynamic languages were not widely used for large system development because they need a large run-time support system. This run-time support increases the storage needs and reduces the execution speeds of programs written in the language. However, the increasing power and reducing cost of computer hardware have made these factors less important.

This means that, for many business applications, these languages can replace imperative programming languages such as C, COBOL and Ada. Java is clearly a

Figure 8.6 High-level languages for prototyping

Language	Type	Application domain
Smalltalk	Object-oriented	Interactive systems
Java	Object-oriented	Interactive systems
Prolog	Logic	Symbolic processing
Lisp	List-based	Symbolic processing

mainstream development language with its roots in C++ yet it incorporates many of the features of Smalltalk such as platform independence and automatic storage management. Java provides many of the advantages of very high-level languages with the rigour and the opportunities for performance optimisation offered by conventional third-generation languages. Many reusable Java components are available so it is clearly a very suitable language for evolutionary prototyping.

Figure 8.6 shows the dynamic languages that are most commonly used for prototype development. When choosing a prototyping language, you should ask a number of questions:

1. *What is the application domain of the problem?* As shown in Figure 8.6, different languages are best suited to different application domains. If you want to prototype applications which involve natural language processing (say), a language such as Lisp or Prolog is more suitable than Java or Smalltalk.
2. *What user interaction is required?* Different languages provide different levels of support for user interaction. Some languages, such as Smalltalk and Java, are well integrated with web browsers while others, such as Prolog, are best suited to text-based interfaces.
3. *What support environment is provided with the language?* Mature support environments with many tools and easy access to reusable components simplify the prototype development process.

Dynamic high-level languages may be used in combination to create a system prototype. Different parts of the system may be programmed in different languages and a communication framework established between the parts. Zave (1989) describes this approach to development in the prototyping of a telephone network system. Four different languages were used: Prolog for database prototyping, Awk (Aho *et al.*, 1988) for billing, CSP (Hoare, 1985) for protocol specification and PAISLey (Zave and Schell, 1986) for performance simulation.

There is never an ideal language for prototyping large systems as different parts of the system are so diverse. The advantage of a mixed-language approach is that the most appropriate language for a logical part of the application can be chosen, thus speeding up prototype development. The disadvantage is that it may be

difficult to establish a communication framework which will allow multiple languages to communicate. The entities used in the different languages are very diverse. Consequently, lengthy code sections may be needed to translate an entity from one language to another.

8.2.2 Database programming

Evolutionary development is now a standard technique for implementing small and medium-sized applications in the business systems domain. The majority of business applications involve manipulating data from a database and producing outputs which involve organising and formatting that data.

To support the development of these applications, all commercial database management systems now support database programming. Database programming is carried out using a specialised language which embeds knowledge of the database and which includes operations that are geared to database manipulation. The language's supporting environment provides tools to support user interface definition, numeric computation and report generation. The term *fourth-generation language* (4GL) is used to refer to both the database programming language and its supporting environment.

Fourth-generation languages are successful because there is a great deal of commonality across data processing applications. In essence, these applications are concerned with updating a database and producing reports from the information in the database. Standard forms are used for input and output. 4GLs are geared towards producing interactive applications which rely on abstracting information from an organisational database, presenting it to end-users on their terminal or workstation and then updating the database with changes made by users. The user interface usually consists of a set of standard forms or a spreadsheet.

The tools which are included in a 4GL environment (Figure 8.7) are:

1. A database query language which is now usually SQL (Date and Darwen, 1997). This may be input directly or generated automatically from forms filled in by an end-user.

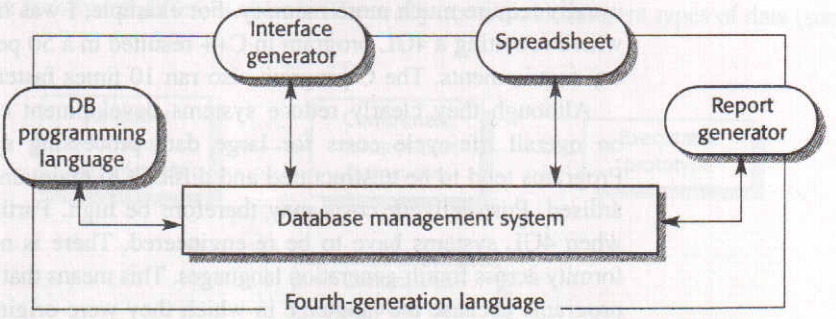

Figure 8.7 Fourth-generation language components

2. An interface generator which is used to create forms for data input and display.
3. A spreadsheet for the analysis and manipulation of numeric information.
4. A report generator which is used to define and create reports from information in the database.

Most business applications rely on structured forms for input and output so 4GLs provide powerful facilities for screen definition and report generation. Screens are often defined as a series of linked forms (in one application we studied, there were 137 different form definitions) so the screen generation system must provide for:

1. *interactive form definition* where the developer defines the fields to be displayed and how these are to be organised;
2. *form linking* where the developer can specify that particular inputs cause further forms to be displayed;
3. *field verification* where the developer defines allowed ranges for values input to form fields.

Most 4GLs now support the development of database interfaces based on World Wide Web browsers. These allow the database to be accessed from anywhere with a valid Internet connection. This reduces training costs and software costs and allows external users to have access to a database. However, the inherent limitations of web browsers and Internet protocols mean that this approach may be unsuitable for systems where very fast, interactive responses are required.

4GL-based development can be used either for evolutionary prototyping or in conjunction with a method-based analysis where system models are used to generate the prototype system. The structure which CASE tools impose on the application and the associated documentation mean that evolutionary prototypes developed using this approach should be more maintainable than manually developed systems. The CASE tools may generate SQL or may generate code in a lower-level language such as COBOL. Forte (1992) describes a number of tools of this type in a brief survey of fourth-generation languages.

While 4GLs are very suitable for prototype development, there are some disadvantages in using them for production systems. Programs written in a 4GL are usually slower than similar programs in conventional programming languages and usually require much more memory. For example, I was involved in an experiment where rewriting a 4GL program in C++ resulted in a 50 per cent reduction in memory requirements. The C program also ran 10 times faster than the 4GL system.

Although they clearly reduce systems development costs, the effect of 4GLs on overall life-cycle costs for large data processing systems is not yet clear. Programs tend to be unstructured and difficult to maintain and 4GLs are not standardised. Post-delivery costs may therefore be high. Particular problems can arise when 4GL systems have to be re-engineered. There is no standardisation or uniformity across fourth-generation languages. This means that users may have to rewrite programs because the language in which they were originally written is obsolete.

8.2.3 Component and application assembly

The time needed to develop a system can be reduced if many parts of that system can be reused rather than designed and implemented. Prototypes can be constructed quickly if you have a set of reusable components and some mechanism to compose these components into systems. The composition mechanism must include control facilities and a mechanism for component communications. This approach is illustrated in Figure 8.8.

Prototyping with reusable components involves developing a system specification by taking account of what reusable components are available. This may mean that some requirements compromises may have to be made. The functionality of the available components may not be a precise fit for the user requirements. However, user requirements are often fairly flexible so, in most cases, this approach can be used for prototype development.

Prototype development with reuse can be supported at two levels:

1. The application level where entire application systems are integrated with the prototype so that their functionality can be shared. For example, if the prototype requires a text processing capability, this can be provided by integrating a standard word processing system. Applications, such as Microsoft Office applications, support application linking.

2. The component level where individual components are integrated within a standard framework to implement the system. This standard framework can be a scripting language which is designed for evolutionary development such as Visual Basic, TCL/TK (Ousterhout, 1994), Python (Lutz, 1996) or Perl (Wall *et al.*, 1996). Alternatively, it can be a more general component integration framework based on CORBA, DCOM or JavaBeans (Sessions, 1997; Orfali and Harkey, 1998; Pope, 1998).

Application reuse gives access to all of the functionality of an application. If the application also provides scripting or tailoring facilities (e.g. Excel macros) these may be used to develop some prototype functionality. A compound document metaphor is helpful to understand this approach to prototype development. The data processed by the prototype system is organised into a compound document that acts as a container for several different objects. These objects contain different types of data (such

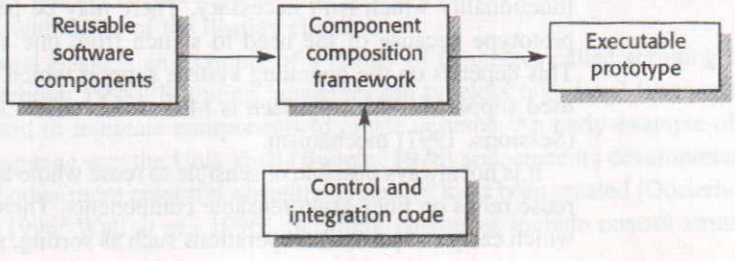

Figure 8.8 Reusable component composition

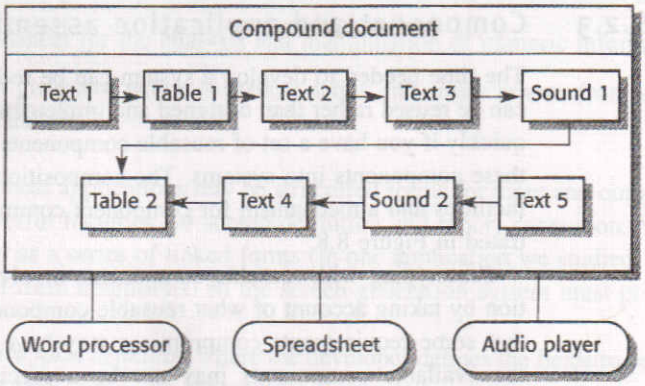

Figure 8.9
Application linking

as a table, a diagram, a form) that can be processed by different applications. Objects are linked and typed so that accessing an object results in the associated application being initiated.

Figure 8.9 illustrates the notion of a prototype system being made up of an compound document that includes text elements, spreadsheet elements and sound files. Text elements are processed by the word processor, tables by the spreadsheet application and the sound files by an audio player. When a system user accesses an object of a particular type, the associated application is called to provide user functionality. For example, when objects of type sound are accessed, the audio player is called to process these objects.

To illustrate the type of prototype which might be developed using this approach, consider the process of requirements management discussed in Chapter 6. A requirements management support system needs a way of capturing requirements, storing these requirements, producing reports, discovering requirements relationships and managing these relationships as traceability tables. This could be prototyped by linking a database (to store requirements), a word processor (to capture requirements and format reports), a spreadsheet (to manage traceability tables) and specially written code to find relationships between the requirements.

The main advantage of this approach is that a lot of prototype functionality can be implemented quickly at a very low cost. If the prototype users are already familiar with the applications making up the prototype, then they do not have to learn how to use new features. However, if they do not know how to use the applications, learning may be difficult, especially as they may be confused by application functionality which isn't necessary. There may be performance problems with the prototype because of the need to switch from one application system to another. This depends on the operating system support which is provided. The most widely used support for this approach is Microsoft's object linking and embedding (OLE) (Sessions, 1997) mechanism.

It is not always possible or sensible to reuse whole applications. Development with reuse relies on finer-grain reusable components. These may be functions or objects which carry out particular operations such as sorting, searching, displaying, etc. The

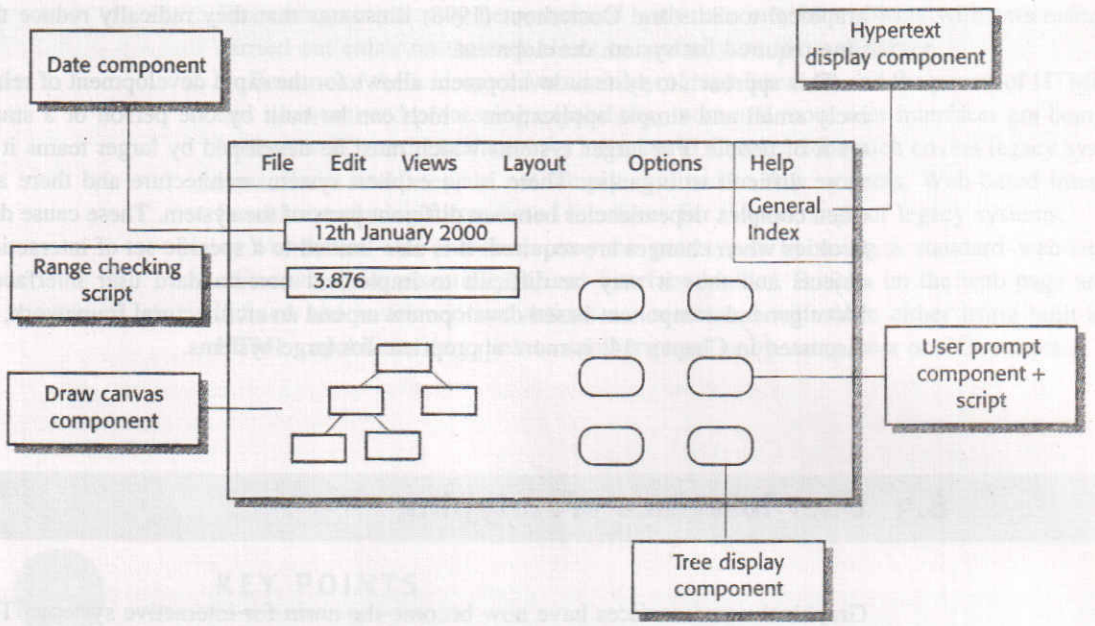

Figure 8.10 Visual programming with reuse

prototype system is developed by defining an overall control structure and then integrating the components with that structure. If components which carry out the required function are not available, special-purpose code is developed and, if appropriate, made available for future reuse.

Visual development systems such as Visual Basic support this reuse-based approach to application development. Application programmers build the system interactively by defining the interface in terms of screens, fields, buttons and menus. These are named, and processing scripts are associated with individual parts of the interface (e.g. a button named 'Simulate'). These scripts may be calls to reusable components, special-purpose code or a mixture of both of these.

I illustrate this approach in Figure 8.10 which shows an application screen including menus along the top, input fields (the white fields on the left of the display), output fields (the shaded field on the left of the display) and buttons which are the rounded rectangles on the right of the display. When these entities are positioned on the display by the prototype construction system, the developer defines which reusable component should be associated with them or writes a program fragment to carry out the required processing. In Figure 8.10, I show the components associated with some of the display elements.

Visual Basic is an example of a family of languages called scripting languages (Ousterhout, 1998). Scripting languages are typeless high-level languages that are designed to integrate components to create systems. An early example of a scripting language was the Unix shell (Bourne, 1978) and, since its development, a number of other more powerful scripting languages have been created (Ousterhout, 1994; Lutz, 1996; Wall *et al.*, 1996). Scripting languages include control structures and

graphical toolkits and Ousterhout (1998) illustrates that they radically reduce the time required for system development.

This approach to system development allows for the rapid development of relatively small and simple applications which can be built by one person or a small team of people. For larger systems which must be developed by larger teams it is more difficult to organise. There is no explicit system architecture and there are often complex dependencies between different parts of the system. These cause difficulties when changes are required. It is also limited to a specific set of interaction objects and thus it may be difficult to implement non-standard user interfaces. More general component-based development around an architectural framework, as discussed in Chapter 14, is more appropriate for large systems.

8.3 User interface prototyping

Graphical user interfaces have now become the norm for interactive systems. The effort involved in specifying, designing and implementing a user interface represents a significant part of application development costs. As discussed in Chapter 15, designers should not impose their view of an acceptable user interface on users. The user must take part in the interface design process. This realisation led to an approach to design called user-centred design (Norman and Draper, 1986) that depends on interface prototyping and user involvement throughout the interface design process.

From a software engineering point of view, prototyping is an essential part of the user interface design process. Because of the dynamic nature of user interfaces, textual descriptions and diagrams are not good enough for expressing the user interface requirements. Therefore, evolutionary prototyping with end-user involvement is the only sensible way to develop graphical user interfaces for software systems.

Interface generators are graphical screen design systems where interface components such as menus, fields, icons and buttons are selected from a menu and positioned on an interface. As I have already discussed, systems of this type are an essential part of a database programming system and Visual Basic (discussed above) has based its standard development technique around such a system. Shneiderman (1998) discusses a number of these systems. Interface generators create a well-structured program generated from an interface specification. The iterations which are an inherent part of evolutionary development do not degrade the software structure and reimplementation is not required.

Millions of people now have access to World Wide Web browsers. These support a page definition language (HTML) which has been extended from a simple text mark-up language to a comprehensive notation for user interface specification. Buttons, fields, forms and tables may be included in web pages as well as multimedia objects giving access to sounds, video and virtual reality displays.

Processing scripts may be associated with user interface objects with processing carried out either on the web client or centrally on the web server.

Because of the widespread availability of web browsers and the power of HTML and its associated processing capabilities, more and more user interfaces are being built as web-based interfaces. As I discuss in Chapter 26, which covers legacy systems, these interfaces are not simply confined to new systems. Web-based interfaces are replacing forms-based interfaces for a wide range of legacy systems.

Web-based user interfaces may be prototyped by using a standard web-site editor which is, essentially, a user interface builder. Entities on the web page are defined and positioned and actions are associated with them either using built-in HTML capabilities (e.g. link to another page) or by using Java or CGI scripts.

KEY POINTS

- A system prototype can be developed to give end-users a concrete impression of the system capabilities. The prototype may therefore help in establishing and validating system requirements.
- As pressure grows for the rapid delivery of software, prototyping is becoming increasingly used as the standard development technique for small and medium-sized systems, especially in the business domain.
- 'Throw-away' prototyping involves developing a prototype to understand the system requirements. In evolutionary prototyping, a prototype evolves through several versions to the final system.
- When implementing a throw-away prototype, you first develop the parts of the system you understand least; in an evolutionary prototype, you develop the parts of the system you understand best.
- Rapid development is important for prototype systems. To deliver a prototype quickly, you may have to leave out some system functionality or relax non-functional constraints such as response speed and reliability.
- Prototyping techniques include the use of very high-level languages, database programming and prototype construction from reusable components. Many prototyping environments support a visual programming approach to development.
- User interfaces should always be developed using prototyping as it is not possible to specify these effectively using a static model. Users should be involved in the evaluation and evolution of the prototype.

FURTHER READING

'Scripting: Higher-level programming for the 21st century'. An overview of scripting languages by the inventor of TCL/TK who discusses the advantages of this approach for rapid application development. (J. K. Ousterhout, *IEEE Computer,* March 1998.)

'Rapid prototyping: Lessons learned'. This is a good summary of the advantages and disadvantages of rapid prototyping of software. It is based on an empirical survey of a number of different types of software system. (V. Scott Gordon and J. Bielman, *IEEE Software*, January 1995.)

'Twenty-two tips for a happier, healthier prototype'. Mostly sound advice on prototyping presented from the perspective of a user interface designer. (James Rudd and Scott Isensee, *ACM Interactions* 1.1, January 1994.)

EXERCISES

8.1 You have been asked to investigate the feasibility of prototyping in the software development process in your organisation. Write a report for your manager discussing the classes of project where prototyping should be used and setting out the expected costs and benefits from using prototyping.

8.2 Explain why, for large systems development, it is recommended that prototypes should be 'throw-away' prototypes.

8.3 What features of languages like Smalltalk and Lisp contribute to their support of rapid prototyping?

8.4 Under what circumstances would you recommend that prototyping should be used as a means of validating system requirements?

8.5 Suggest difficulties that might arise when prototyping real-time embedded computer systems.

8.6 A software manager is involved in a project development of a software design support system that supports the translation of software requirements to a formal software specification. Comment on the advantages and disadvantages of the following development strategies:

(a) Develop a throw-away prototype using a prototyping language such as Smalltalk. Evaluate this prototype and then review requirements. Develop the final system using C.

(b) Develop the system from the existing requirements using Java and then modify it to adapt to any changed user requirements.

(c) Develop the system using evolutionary prototyping using a prototyping language such as Smalltalk. Modify the system according to the user's requests and deliver the modified prototype.

8.7 Discuss prototyping using reusable components and suggest problems which may arise using this approach. What is the most effective way to specify reusable components?

8.8 What are the advantages and disadvantages of using Microsoft's OLE mechanism for rapid application development?

8.9 You have been asked by a charity to prototype a system that keeps track of all donations they have received. This system has to maintain the names and addresses of donors, their particular interests, the amount donated, and when it was donated. If the donation is over a certain amount, the donor may attach conditions to the donation (e.g. it must be spent on a particular project) and the system must keep track of these donations and how they were spent. Discuss how you would prototype this system bearing in mind that the charity has a mixture of paid workers and volunteers. Many of the volunteers are retirees who have had little or no computer experience.

8.10 You have developed a throw-away prototype system for a client who is very happy with it. However, she suggests that there is no need to develop another system but that you should deliver the prototype and offers an excellent price for the system. You know that there may be future problems with maintaining the system. Discuss how you might respond to this customer.