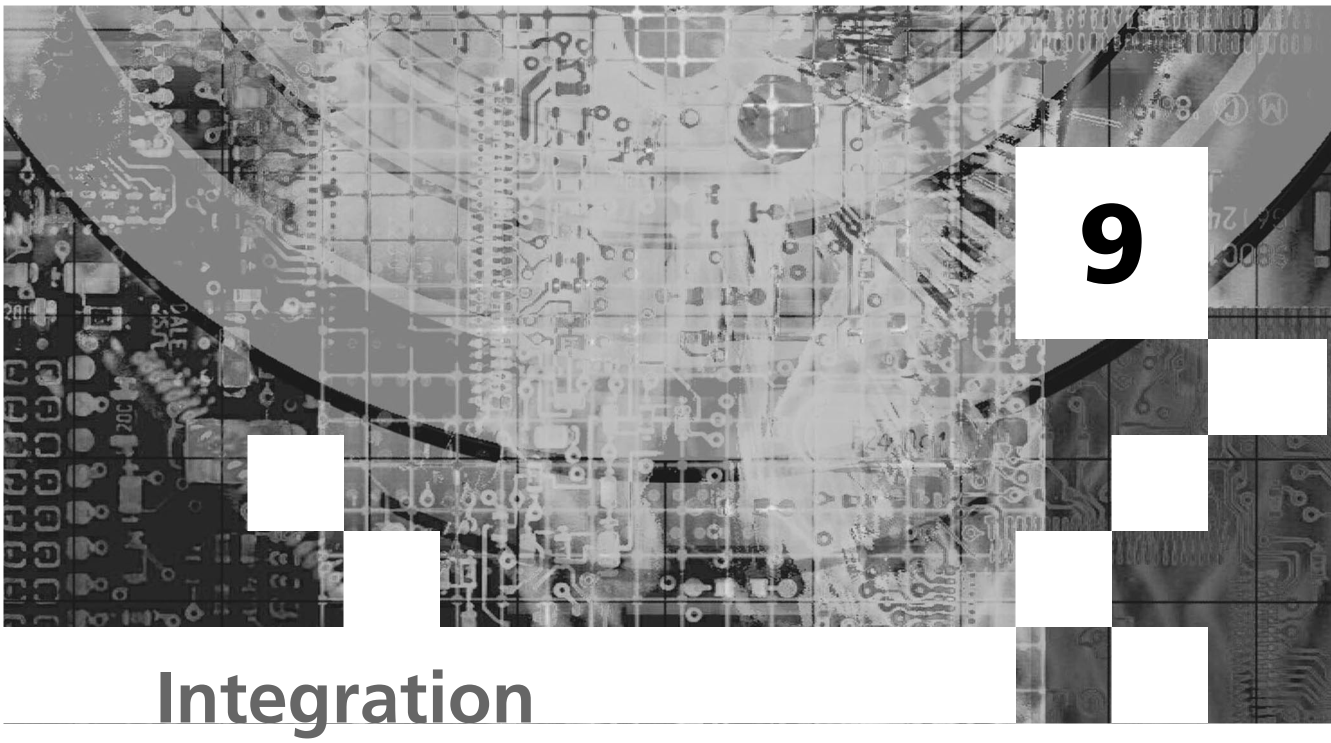


9 Integration

Project manager's responsibilities

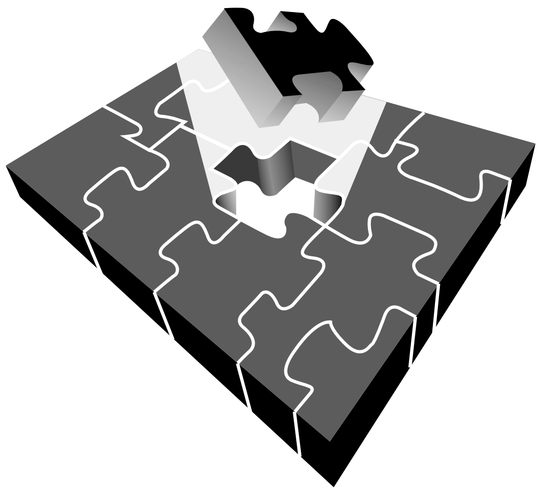
- To assemble the necessary software engineering team to implement the application
- To ensure that the project definition is in a form that can be implemented in software
- To monitor the development of the programming, and liaise between the client and programming team over changes and misunderstandings
- To define the testing to be carried out, and oversee that process
- To understand the basics of programming logic so as to better understand the problems facing the programmers

New media fusion or confusion?

Interactive media can be said to gather the best, and the worst, aspects of both computing and the audiovisual industry, and they have to be fused together in the core stage of multimedia development. This is the integration of the application, whether it is getting the parts of a website to work together or producing an iTV programme, a kiosk or CD-ROM. Here the underlying computer software is applied to the assets and/or the data, and this is often the first time that anyone, including the designer, sees the jigsaw puzzle fitting together. This chapter explores this integration and looks at programming as it applies to websites and offline projects: for simplicity I'll just call them all 'applications'.

The writing of HTML for a website is not often regarded as programming, especially because it is often carried out by another specialist, such as a graphics artist. However, JavaScript or CGI applications on the Web server certainly constitute programming, and can be just about as arcane as multimedia gets. So once you get beyond HTML you soon run into programming.

Managing a team in software development has much in common with the general principles covered in Chapter 14 of Book 1, *Team management principles*. The software team involved in an application can range from a single person upwards, although in multimedia software teams are often no larger than four or five. If you have a single person working on your project it will be beneficial for that person to have contact with other programmers so that any sticky problems can be sorted out. Within a team you may wish to have one person who manages the team's work. Software is often written on a modular basis, with separate but self-contained parts of the application being written by different people. It is often necessary for someone with software knowledge to distribute the work and make sure the modules fit together.



The choice of people to program your project will depend on how it will be carried out. Different software engineers will have experience of using different languages and tools and working in different environments. For example, dynamic websites need software engineers experienced in server-side or back-end programming while a kiosk is likely to need someone with experience of a package like Director. Interactive television might need C++ skills, or HTML/XML depending on the particular platform. WAP uses a markup language similar to HTML and written in XML. So people with HTML, XML, Java or C++ experience can fit into teams in many areas. Ideally you will have enough experience, or be able to get suitably impartial advice, to choose the environment and therefore know what kind of programmer to recruit.

A second part of managing software involves the specifying of what needs to be done and ensuring that it is carried out. Taking advice from a programmer during the early stages of the design can avoid problems later and can help you to describe the application in ways that a programmer can clearly understand. When specifying an application there will be input from every part of your team and from your client. For a consumer product your client's input could be augmented by information on what customers want, but the basic management principle remains the same.

The purpose of this chapter is to give an overview of the software aspects of websites and multimedia, but it does not set out to show how to actually carry out the programming. Just as in the audio and video chapters, this chapter aims to help non-programming specialists to understand the processes and problems that arise in multimedia software development.

A computer program is a series of operations and decisions. Something happens as a result of the user carrying out an action, and the software will respond. It is the task of the designer to decide what that response will be, and of the software engineer, or programmer, to implement this as code. The software engineer will write the software that carries out this integration by using either an authoring tool or a programming language.

Authoring versus programming

There is no hard-and-fast rule about what constitutes authoring as distinct from programming. Packages with which you can write multimedia applications range from graphical packages such as Director and ToolBook to Java and C++. With the first you can build basic interactive structures with a few clicks of the mouse button and a few pulled-down menus, through scripting and mark-up languages such as HTML, which is not unlike English, to the hard stuff, such as Java and C++. In general the versatility and performance of your application will increase as you move towards a full-blown programming language such as C++, but it will be more difficult to implement.

Offline applications have traditionally been a homogenous environment for programming in that many of them were written using a package like

Director and it was rare to need to go outside this particular environment. A website is now a heterogeneous environment, needing a working knowledge of more than just HTML. Dynamic HTML (DHTML) goes further by allowing sections of a web page to be defined and labelled so that they can be dynamically changed or moved around under JavaScript control. Cascading Style Sheets (CSS) provide a site-wide mechanism for controlling the look of a site. JavaScript is a programming language with great power and can work with DHTML and CSS to manipulate web pages. Currently, an increasing number of websites are moving away from HTML towards XML and the combination of XML to describe the page content with CSS to define how it should be displayed. Unfortunately the implementations of these technologies by the makers of web browsers are different. In one key area Netscape and Internet Explorer have used a different version of what is called the Document Object Model to describe the elements of a web page which makes it difficult to write JavaScript that will work successfully across all browsers.

Even if you program in a sophisticated language like C++ or Java, it doesn't mean that you have to write everything from scratch. You will have access to libraries of self-contained code routines or objects. You might write these and reuse them from project to project, they might be a constant part of the operating system you are writing for or they might be libraries of code you buy to enable to program a particular system. (See a discussion on copyright in computer code in Book 1 Chapter 15.)

The objects in the libraries will have a known and probably simple function and will come with instructions telling you how to call the routine and pass parameters to it and what it will do and/or return to you after it has run. These objects are often like 'black boxes' in that you don't need to know how they do their job, you just need to know exactly what they do and how to instruct (program) them.

An authoring package often lends its own look and feel to the application, and experienced authors can sometimes look at a program and say 'Oh, that was developed using Macromedia Director' or 'That's a typical ToolBook application'. This is partly the result of early users of such tools making good use of the facilities the tools provide – a package-led approach. Using authoring packages for web pages has an extra aspect, in that often the HTML they produce is quirky in some way, even though it usually still does the job. (Also some packages leave their footprints in the HTML, either in the creator meta tag at the top of the web page source or because they name JavaScript functions in a certain way.)

As time has gone by, the desired functionality has become more of a driving force than the obvious abilities of the tools. It must be said, however, that authoring tools do have their limits, especially when it comes to performance of offline applications.

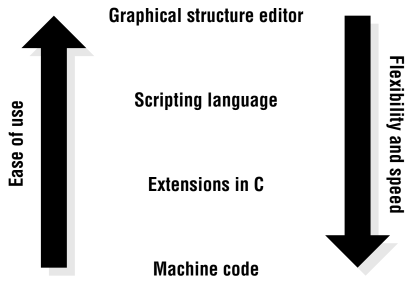
So, how do you choose between the different options? In the end some kind of risk–benefit analysis needs to be done to compare your options, but there are a couple of underlying factors that you need to take into account.

As a general rule, the lower the level of coding you undertake, the longer it will take. This is partly because the code will take longer to write, but it will also be more prone to bugs that are difficult to track down. To counter this, if you have software engineers working in your team who have experience of lower-level coding then they may program very quickly. You, or your software team, should be building up a library of objects, or routines for your programming. This is particularly true if you are working in Java or C++. As we've mentioned earlier, you can incorporate these libraries in future projects as well. This is the main reason why it is a good idea for you to only license your code to your clients rather than grant them all rights; otherwise you will find it difficult to make use of your basic routines from earlier projects.

Besides the inherent abilities of the team at your disposal, you will also be judging the requirements of the particular application. In some cases it may demand low-level coding, for instance if there are any gaming elements, but for less responsive and more asset-led applications an authoring tool may be very appropriate.

The authoring tree

A good graphical authoring package or structure editor will have an underlying scripting language, and a good scripting language will allow you to build new commands with a lower-level language such as C. (The web-authoring package Dreamweaver allows you to write extensions to itself in JavaScript for example.) At the bottom of this pile, rather like the court of ultimate appeal, the low-level language could call routines in machine code for speed. Machine code written as part of an application is common in games but much less so in multimedia.



A graphical structure editor, or similar, lets you define the flow of an application's logic by drawing it on the screen in some way, often just like a flow chart in traditional logic. Examples of this kind of authoring package include Icon Author and Authorware.

Alternatively the flow could be predetermined, as in a card-based hyperlinked system, where you define hot-spots or buttons and then have a new card that is displayed as a result of activating the button. The buttons can also trigger other things besides jumps to other cards, and you can trigger events on opening or closing cards as well. Examples of this kind of system include ToolBook and HyperCard.

A third kind of authoring system uses a time line onto which you put events. The hyperlinks in this kind of system jump around on the time line. Macromedia Director is an authoring system of this kind.

Although these are in fact three different kinds of authoring system, what they have in common is that you can build a simple application or define the outline of a more complex one without typing in any programming commands. In almost every case, these authoring tools have scripting languages underneath, such as Lingo with Director and Hypertalk with HyperCard.

There is no hard-and-fast definition of an authoring or scripting language as opposed to a programming language. One possible definition is that a scripting language is a special type of computer language in that it usually appears to be in English. HTML is sometimes considered to be a scripting language, although it actually sets out to mark up text to say how it should be displayed. As HTML has evolved to allow designers to control more of the look of a page it has become more of a program and less of a mark-up. (However, since HTML has no true variables or structures to allow conditional looping and testing it falls outside any real definition of a computer programming language.) Scripting languages can also offer intuitive variables like ‘it’ so that when you ‘get the time’ the value of ‘the time’ is held in a variable automatically called ‘it’. Then when the next command is ‘show it’ the value of the time will be displayed. Since the script says ‘get the time, show it’, it reads like English.

The definition falls down partly because well-written code in computer languages can sometimes be as relatively easy to read as English. There are even poems written in Perl which constitute ‘legal’ programs. Also C++, which is a very powerful programming language, has a variable ‘this’, which always refers to the current object. JavaScript and Perl are two languages used on the Web. JavaScript usually runs in the browser and is often embedded in the HTML, whereas Perl always runs on the server. Both these languages are often referred to as producing scripts rather than code, but this just serves to illustrate how blurred the boundaries are since their syntax can be decidedly cryptic at times.

Another definition is that a scripting language will only be able to carry out a defined range of tasks, whereas a programming language is versatile. This one is closer to the truth but still not accurate. As long as a language can carry out basic arithmetic and logic it can, in theory, be applied to any task – even being used to write itself, just as C can be used to write a C compiler. But that may be the key. In practice, most scripting languages are good at some tasks but very inefficient in others.

To help with their core task, scripting languages for multimedia will have built-in support for sound and video, which would have to be specially programmed in a more general-purpose programming language.

Because of the hyperlinked nature of the HTML documents used on the World Wide Web, you could argue that HTML is a scripting language, and the kind of programs that read HTML (Web browsers) can handle pictures, sound and movies. It is possible to use HTML as a simple form of multimedia authoring on CD-ROM (as we did on the CD in earlier editions of this book and has become increasingly popular over the last few years). HTML is designed to work in a distributed system – the Internet – and distributed multimedia is becoming increasingly important.

In an authoring environment you, as author, do not really have any control over the way your program interacts with the computer’s memory, filing system and so on. This can result in problems if the authoring package you are using has any faults in the way it interacts with the computer. One example is where the program takes some memory to carry out a task but does not correctly free that memory after use. In this way it is possible for a program to very slowly eat into free memory until the computer runs out of usable memory and crashes. This kind of error, known as a memory leak, is extremely difficult to track down, and can be the cause of inexplicable crashes that also seem to be unrepeatable. In this case, using a utility program to monitor the computer can help. If you are working in a computer language that allows direct control of such things, such as C, then you have to take care of memory usage yourself.

JavaScript has its own issues with memory. The JavaScript in a web page has to run in the available browser memory and it is entirely possible for several JavaScript routines to be active simultaneously depending on how the website is designed. Browsers behave unpredictably under such duress.

A final analogy for programming versus authoring: you might be able to use a screwdriver (a low-level and versatile tool) to dig a hole in the street, but would you want to do so? The jackhammer or pneumatic drill (a rather blunt analogue of the authoring tool) is better suited to the job, but less versatile than the screwdriver. Conversely, think about trying to turn a screw with a jackhammer.

Stages

There is some confusion as to the stages of software development, since some of the terms mean different things to different people. In practice, because of the diverse backgrounds of people working in multimedia, a strict software-oriented development cycle might not be followed, or indeed be appropriate, as has been discussed earlier in Table 2.1 (Book 1 Chapter 2, *Multimedia and project management*). It is up to the development team as to how they manage their programming, but some clients will expect some of the more formal elements of software design. To help understand this, what follows is a brief outline of a more software-oriented procedure. Although this is primarily appropriate for an offline application don't forget that a dynamic website also behaves as a single application when the client (browser) and server-side functions are taken together.

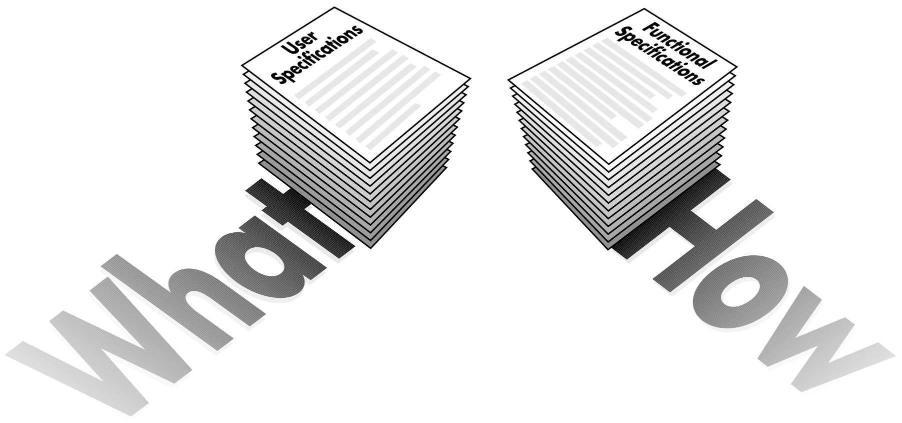
In addition, the quality standards (such as ISO 9000) that can be applied to any process lay down definite meanings. One common software approach is to write a user requirements document and a functional specification. The proposal to the client may well have covered some aspects of this, since the application structure might have been outlined. Ideally someone from the software team would have been involved in this process, or else the structure would have been based on previous work and therefore would be, to a large extent, tried and tested.

User requirements

'User requirements' is an ambiguous expression, and if it is mentioned, you will need to be sure of what is actually meant. There are two meanings, both of which are useful, and you might consider including both in the software process.

The first meaning is 'What are the needs of the user that the application can satisfy?' In this way the user requirement of a word processor is basically the ability to write and manipulate text, and this kind of information follows from the scoping of the project when the needs and objectives of the user are determined.

This is a very general interpretation, which allows the design team to lay down the functionality of the software in a general way. Sometimes this



stage is called a user specification, and in order to help a potential client or sponsor understand what the application will do, it could include short scenarios that describe (hypothetically) what it would be like to use the application.

The second meaning is 'What does the user require of the application?', which is a much more specific question. Here the answers are things such as 'a user-friendly interface', 'spell checking', and 'automatic saving of work on a regular basis'. It may be that your client will have very specific requirements that the complete application must fulfil, and a strict user requirements list like this is a good way of expressing those. The software team uses the list as input saying what needs to be done. It is their task to work out how to do it, which leads to the functional specification.

Functional specification

Whereas the user requirements outline what needs to be done, the functional specification gives much more detail about how a task will be carried out. At its fullest a functional specification will list the outcome of every action carried out by the user, and will say how that is to be achieved.

With a website some things are taken for granted: the way the links operate and the graphics are displayed are outside your control since the browser does that. However, the client and the developer need to agree on the structure of the site and the way it looks. So a specification for a simple static website might concentrate on the look of the graphics – perhaps with template designs for page layouts – and the hierarchical structure of the site. Once a website moves beyond HTML then a more formal functional specification becomes a possibility. This can happen because the user interface is going to make use of JavaScript to improve the user experience. Or the site might be very large and dynamic with online shopping, perhaps something like a catalogue, with many pages that have a standardized lay-out. In such cases the website might be built dynamically by a database. The database will organize the site content, and the responses to queries will be output as HTML or HTML and JavaScript so that the browser receives what seem to be ordinary web pages.

For a major software project (and this would include a big e-commerce website, for example), the functional specification will be a very large document, and will be carefully researched and written. It can take many months to write the specification, and it is the document that defines the application. If clients have a software background, they may ask for such a document and often the developer will insist on one in any case. Otherwise, the programmers will perform the same sort of task if not necessarily on the same scale. In this case the clients may not be given the programming specification as part of the documentation because its use to them without a programming background would be minimal. If clients have requested a functional specification, then once this document is agreed and signed off, any changes have to be similarly discussed, agreed and signed off as change requests. A change request can then be evaluated in terms of its impact on time, quality and budget as discussed in Book 1 Chapter 5, *Contract issues 1*.

There is a basic difficulty in applying a strict functional specification of this type to multimedia, which is why its use is intermittent in the industry. This results partly from the audiovisual nature of multimedia. A lot of the content of the application lies in the assets, and they can be very difficult to specify since, at the start, they will not have been researched. Also, time-based media such as movies are notoriously difficult to specify since the performance of the integrated application depends on so many factors, including the exact nature of the time-based asset itself. We are not yet at the stage in multimedia where our clients and viewers can take as read the quality thresholds that apply to assets in the way they can for radio or television. The vague but widely understood concept of broadcast quality cannot yet automatically apply to multimedia simply because of the variety of display systems on people's desks and restrictions of delivery. Instead, custom and practice and easily available examples of quality in other applications and websites help to demonstrate that the best quality possible in the circumstances is being achieved.

A second problem with the functional specification lies in the difficulty of writing down what is a very dynamic process. For this reason many developers use prototypes, or demonstrator applications sometimes also called animatics, to define the task to be carried out. Even then there is an acceptance that not everything intended can be achieved. Sometimes for multimedia the only accurate definition of an application will be the application itself.

Finally, a recent trend in software design has been to forgo a functional specification altogether. With object-oriented programming, where distinct modules of the program are defined carefully so that they operate as independent objects and just communicate with each other, the important thing to define is the way the program will work. The exact detail of how the objects are coded is left to the software engineers.

For these reasons any kind of functional specification, and also the user requirements, are likely to be internal rather than external documents in a multimedia development. Their primary purpose is to help the software team to build the application rather than to define the application for the client. As a result the documents may even be called something completely different but would fulfil broadly the same purpose.

Alpha and omega

The actual development, as distinct from the talking about the development, also has names. Again, there is some variance as to what the terms really mean, but this is one set of meanings which is possibly more common with offline projects than with online. Software people will sometimes refer to stages of a project in terms such as pre-alpha, alpha, beta, and golden master. A wider approach to testing than a pure software approach is discussed in Book 1 Chapter 11, *Testing*. Using this as a reference, the point at which you would put your project out for external testing is what would be referred to as a beta version by software-oriented developers and can happen with a website just as with any offline application.

An alpha is more for testing internally in the team, and you might not even show it to the client. To extend the meaning of alpha into multimedia you might say that an alpha is structurally complete but does not have all its content in place, only enough to test functionality while you are waiting for the complete content. With a website this stage might be a graphical mockup or static prototype, done in something like PhotoShop, to show the design, after which and on approval, the HTML is done to build the page. This is part of the type of testing referred to as developmental testing in Book 1 Chapter 11.

In offline projects, the golden master is the one that is actually going to be sent for replication and distribution. The equivalent stage for a website is the point at which the complete site is ready to go online. It is fully tested and has survived all the testing you, and possibly your client, has thrown at it. The project is signed off or gets final acceptance and might even be packaged and sent off to the client's IT people to go onto their web server.

As the software team gets bigger, the difficulties of keeping track of the development increase. In new media this problem of version control or version tracking, as this is called, increases as well because most of your assets will be going through changes too. As part of your version control you will need to devise a numbering scheme for everything you do, including the documentation. Here are some suggestions:

- Anything that is unfinished has a version number smaller than one, for example 0.3 or 0.99 (for 'almost there'). You can have more numbers after the decimal point to show very small changes but this is a rather subjective concept.

- A more complex numbering system, such as 1.2 B 24 would denote that this is the beta version of version 1.2 of the specification, and this is the 24th build of the application. The term 'build' implies a compiling process or some process that takes the software in one form and turns it into another.
- An odd or an even number beyond the decimal point could indicate whether the change from the last number was due to new functionality or a bug fix.

In a modular piece of software a version number should be given to every module, especially if different people are writing different modules. Use comments in code freely to explain what is going on, including the basics of what the module does. Images built as montages of other images will also require version numbers.

Bugs

Computers are, at best, very stupid but very logical. They never know what you mean, they only know what you say in your programs, and this can be extremely frustrating when something goes wrong with a program and it has to be investigated.

Legend has it that the use of the word 'bug', meaning a problem in a computer program, dates from Admiral Grace Hopper, who was one of the very early computer pioneers (and sometimes credited with inventing the software computer program back in the days when computers were programmed by changing the wiring). A big computer failed at 1545 on 9 September 1947, and the cause was an insect (bug) which had crawled onto a logic module and died as a result of the heat, voltage or just being trapped by a relay. The legend is only partly true, since engineers have been finding technical 'bugs' in things since the nineteenth century (the modern use of the term is reputed to have originated in telegraphy) and the log entry only says that this was the 'first actual case of a bug being found'. Hopper's insect bug (a moth) did, however, die its legendary death and now lies taped to her log book in the Smithsonian. You can see the log page, complete with moth attached, at http://www.waterholes.com/~dennette/1996/hopper/bug.htm.

There is a saying in programming, 'There is always one more bug', and unfortunately the reliability of complex systems decreases geometrically with the number of component parts. Since a computer program does not wear out like a car engine, its reliability is better defined as correct operation under all conditions. Things the user is not supposed to do – such as illegal keystrokes – should be catered for during software development; but reliability tends to decrease with the increasing complexity of

the program. Even changing the delivery medium can show up a bug. As an example consider an application developed under Windows that is to be delivered on CD-ROM. Windows now allows very long file names, and it is a temptation to use them, but a Windows CD-ROM that uses what are called the Joliet extensions to allow long file names still allows only up to 31 characters in the name, so the program that worked on a hard disk may not work on the CD-ROM. It now seems to be accepted that a complex computer program can never be tested for absolutely every eventuality.

This can be difficult for clients to understand. If they come from an area where it seems that absolute quality is achievable they might overreact to a bug. The onus is on the software team and the project manager to test as thoroughly as possible, bearing in mind cost and time constraints.

One option is to allow the client to use the software 'in anger' for some time and delay final acceptance until that period is finished. If you are going to do this then it is vital that the schedule and cost allow for it and that this has been agreed up front, not as an afterthought. See Book 1 Chapter 11, *Testing*, for how to specify a full testing strategy.

If it is any consolation, even long-established software can have bugs, some known and some unknown. One very common microprocessor, which was for a long time a mainstay of 8-bit computing, had a low-level addressing bug that was never fixed. If a bug is known about, and software works around the bug, it is possible that the bug will never be fixed because this would affect the existing programs that are working around the bug.

With a web page, the difficulty in testing lies in the many different browsers and machine combinations that people will use. A regular survey of visitors to the Browser Watch website turns up so many differently identified browsers visiting the site (this information is sent to the server with the page request) that four different web pages are needed to list them. This survey shows that there are still people out there using version 2 browsers or earlier – but not many of them fortunately.

Of course you can test and validate the syntax of your web page. Programs such as WebLint (which runs under Perl) will do this for you, and track down unbalanced tags (opened but not closed and vice versa) and the like: there are others, and equivalent validators for JavaScript. If you are using a web authoring tool like Dreamweaver then this should prevent you generating bad code and to a great extent will validate your existing code but, because Dreamweaver doesn't mess with your existing code unless it is incorrect, you can sometimes find problems which are more of an idiosyncrasy than a bug.

Sometimes a bug can appear out of a clear blue sky. The author suffered from crashes of a website that suddenly started for no apparent reason, only when using Netscape, and could not be replicated on the local machine, only over the network. This was eventually traced to a background graphic that had somehow become corrupted and was only affecting Netscape under certain circumstances. The graphic was replaced and the bug went away. Bugs like that are hard to find and, as usual, require a complex process of trial and error testing using a staged process of elimination. This process isn't helped with websites by the various caches in the system that can lead to you seeing a different version of a page to the one you think. Putting a version number in a comment at the top of a page can help.

It can be difficult to test CGI programs because they need to run on the server and get data such as environment variables over the network. Environment variables can be passed to a CGI program so that the program can act on them. This makes it difficult to test them offline prior to installation. Often when a CGI has a bug and does not return the correct result you will not see anything on the browser because the server does not allow the CGI to return anything at all. For this reason some servers allow you to run your CGIs in a debugging mode, which allows you to see remote results clearly. This might be done by prefixing the program name with something that identifies it as a debugging run.

A further consideration is that the Web itself suffers from the equivalent of bugs. Some days it slows down and very occasionally it suffers from a major problem. The server could malfunction: it could have crashed because of some external event you could not control, or your CGI program may have seized up, perhaps waiting for a key-press acknowledgement that will never come. If you have an ongoing responsibility for a site it is a good idea to have someone check it every morning or set up an automatic monitoring system. That way, if the client telephones you can already be on the case.

For more on Testing you should refer to Book 1 Chapter 11, *Testing*.

The demo factor

In many organizations and projects there will be pressures to demonstrate websites or offline applications. This could be for the client, or other clients, or the board, or at a trade show. The demo factor brings two major implications to the development of the application.

First, and this is a negative factor, the timing is likely to be such that you will have to assemble a special version for demonstration. This is very likely to be the case if the demo is taking place early in the development cycle. If it is a networked application you may have to produce a stand-alone version, or you may have to put it on the Internet with password protection.

Second, and this is a positive factor, a demo can be a very good bug finder. There is something about letting a member of the company management demonstrate the software that brings the bugs out from the woodwork. Of course this means that the demo should be 'scripted' so that nothing untoward happens. Unfortunately people doing demonstrations do not always follow instructions. The author was involved in a project where the demonstrator was warned that under no circumstances should he do a certain thing because it was known to crash the program. He, of course, did just that – at the beginning of his demonstration to the chairman of the multinational company for which he worked.

However, this is a lesson that you learn very quickly the hard way, and it is a lot easier to lose a client than to gain one. The best approach is not to be press-ganged into giving demos too early.

The only other solution is to make sure that even a demo is solid enough to continue working and not crash completely. Do not have buttons that call a routine that crashes; rather, make them 'blind' and not do anything. It takes minutes to restart most multimedia systems.

The jigsaw

Software engineering is becoming more and more like a mix-and-match jigsaw puzzle. Programs are made up of objects that interact with each other and can be reused in other applications. Multimedia applications will have three main sections that need to be fitted together:

- The **user interface** is the means by which the user controls the program. It may or may not include assets of its own.
- The **program logic** runs behind the user interface and carries out the tasks the user interface requests, and will possibly have an agenda of its own if, for example, the application includes any simulations.

- The **assets** are the textual, graphical and audiovisual components of the program that the program logic will choose and activate, usually on the instructions of the user interface. In a database-driven application there could easily be tens of thousands of these.

Even if the application does not easily break down into these three sections, there are reasons why you might wish to separate them from a logical point of view. The user interface may run remotely from the core program logic in a networked application. A web page is essentially a user interface that runs on a web browser. Assets will need to be proofread or otherwise checked, which is their equivalent of debugging. Replacing assets on a CD-ROM is easy if they are not inexorably entwined in the software, unless of course you have a CD-ROM that has just been shipped out to a thousand customers. Replacing assets on a web page is fairly trivial at any time. Program logic is less platform dependent than the user interface and so is easier to port from one platform to another.

Risky business

There is a wish in most software teams for them to be exploring new territory and not continuously reworking the same old routines time and time again. It is considered a matter of pride to be asked to beta-test some new software tool or operating system extension. When surfing the Web you will often come across websites that require either a just-released browser or some plug-in or other. In many cases Web browser plug-ins – and this is also true of Java – provide tiny run-time environments in which the small programs can run, sometimes called virtual machines if they are particularly versatile. In fact these small programs have even been given a name: applets. Ironically, as you move further and further away from HTML, your online programming becomes more and more like programming for offline multimedia.

'Pushing the envelope' can be a dangerous business. The envelope is your current level of expertise, and when you push the envelope you try something newer and more challenging. To build an application you need to be able to plan and move forward through the development cycle, and if the sands shift beneath you because of bugs in the tools or changes in their functionality then you are in trouble. There are few things worse in computing than chasing a program bug for months only to find eventually that the problem lies in the operating system, or the language, or some other third-party software such as a low-level driver. Code that links (or glues together) two different pieces of software can be especially awkward to debug. It is not unusual for two companies whose software will not work together to blame each other for the problem.

Of course there always has to be a first time to use a new tool, so you can never be completely sure. The message is to think very carefully of the implications of using something new, and to allow for extra time for the learning curve in your planning. The time when you use a new version of a tried and trusted tool or of the operating system can also be problematic, and as a general rule upgrading should not be carried out in mid-project unless there is no alternative.

It is also possible that using a previously untried feature of familiar software can cause difficulties. Unfortunately it is not always possible to rely on documentation since a complex program, by definition, is very difficult to document fully. And as already mentioned, there can also be bugs in long-established software and even the operating system or processor.

The link between software and the client

The project manager's dilemma.

More than in any other area of new development, the project manager or producer has to be able to act both as a buffer and as a translator between the client and the software team. As project manager you have to trust your team to truly represent the status of the software. Sometimes they will give you what seems completely counter-intuitive advice as a result of some esoteric way the browser, compiler or authoring tool works or because of a known problem with the delivery system. The risk is that the client, if not familiar with software engineering and programming, will sometimes think that there is no substance to the real problems you will face. You are the liaison between the team and the client, and you need to be able to understand the problems the software hits in order to explain such things. There is a fine line between explaining a complicated problem to clients and blinding them with science, and the position of that line depends a great deal on your relationship with the client.

THEORY INTO PRACTICE 9

Talk to people you know who have programmed multimedia applications and ask them what they liked and disliked about the process and about the tools and languages they used. You should take any opportunity you have to try new tools, and you will find demonstration examples of some on the Internet.

Summary

- Integration is the core stage of multimedia development, where assets and software are fused together to make the application.
- Choosing the software package, programming language or authoring system is key to this stage. Multimedia authoring systems will have built-in support for different kinds of asset.
- You will need to define names and numbering conventions for the different stages of your development and testing.
- Testing and fixing bugs in complex code will be time-consuming and it is essential to avoid the possibility of bugs in tools and operating system routines as they will confuse the testing.
- Making use of browser plug-ins and Java results in a programming process more like offline multimedia than online web page design.
- 'Pushing the envelope' is risky.
- The software development in multimedia is likely to be the least understood part of the process for a client.

Recommended reading

Apple Computer Inc. (1987). *The Apple Macintosh Human Interface Guidelines*. Reading, MA: Addison-Wesley or see http://developer.apple.com/techpubs/mac/HIGuidelines/HIGuidelines-2.html for a downloadable version.

Niedernst J (1999). *Web Design in a Nutshell*. Sebastopol, CA: O'Reilly & Associates

Sebesta R.W. (2001). *Programming the World Wide Web*. Reading, MA: Addison-Wesley

Spainhour S. and Eckstein R. (1999). *Webmaster in a Nutshell*, 2nd edn. Sebastopol, CA: O'Reilly & Associates

Vaughan T. (2001). *Multimedia: Making it Work*, 5th edn. Berkeley, CA: Osborne McGraw-Hill

The BrowserWatch website is at http://browserwatch.internet.com/ (Note: there is no www in the URL)