ADVANCE EDUCATION

PROVIDING ADVANCED EDUCATIONAL TECHNOLOGY & SYSTEMS

http://ae.maths.uwa.edu.au/

ae@maths.uwa.edu.au

FLYINGFISH V1.51

DOCUMENTATION · APRIL 2000

Dr Nathan Scott

Visit the Advance Education website to download the latest versions of the executables for your platform, and get examples, help, software license costs, FAQs and much more.

The Jellyfish Tutorial Environment would not exist except for the work of Dr Kevin Judd and Prof. Brian Stone. The text of this manual was written by Dr Nathan Scott but the software described is really a joint effort over many years.

Contents

1. Introduction

1.1 What is FlyingFish?

Essentially it is a tutorial environment or system. It allows staff to set a sequence of problems (which can take a wide range of forms) and also deadlines and marking strategies. It also allows students to log in using a Web browser in order to solve the problems. At any time staff can see how students have progressed, and marks can be awarded.

There are many software packages available today that provide these services. So why should you use FlyingFish? Because it is more powerful and flexible than any current competitor. Because it is available on all the main computer platforms. Because the file structure and format used by the system are clear, logical, and will not go out of date. Because it was developed over many years by University teachers who happen to also be programmers, rather than by programmers trying to follow the ideas of teachers. Because the system and those who use it have won many awards for both excellence and innovation in teaching.

1.2 Why is it called "FlyingFish"?

The 1997 version of this system was called "Starfish" because it worked with a Web server called WebSTAR™[1]. However we had to change the name because it was too similar to the name of several existing commercial ventures[2]. The name we initially chose was "Jellyfish" but it never really caught on. Jellyfish are just not very inspiring. This release is called FlyingFish because it is faster and smaller than its predecessors.

The name "Jellyfish" is now used for a set of standards for course materials, developed collaboratively by Dr Kevin Judd and Dr Nathan Scott of UWA.

It is useful to have a name for the tutorial system since it is easier to say "The FlyingFish will respond like this..." than "The server will...". The actual name is not terribly important since it is rarely seen by students.

1.3 Where can it be used?

At the time of writing, problem sets have been developed for:

- Engineering Dynamics, Statics and Thermodynamics;
- First-year University Calculus, Statistics and Linear Algebra;
- Second-year Pathology (identification of cancerous tissue).

Some of these problem sets are quite extensive and represent an entire year of tutorial problems. The system is useful in all of these different subjects because it allows so many different kinds of *problem type*. Here are some examples:

- Problems that provide randomised numerical parameters to the student, and require a numerical answer with units;
- Problems that pose a problem in algebraic form and/or require an expression as an answer;

---

1 A product of the Quarterdeck Corporation, http://www.starnine.com/

2 E.g. http://www.starfish.com/

- Problems that present an image as well as some markers to be “dragged” onto the image;
- Problems that allow the student to sketch a graph as the response; and of course
- Multiple-choice questions.

New problem sets are generated when an enthusiastic person either adapts existing problem types, or creates a whole new problem type, and then generates an appropriate sequence of questions for his or her students. A problem set can consist of any number of problems, of any type, in any order. Problems can also be linked together so that they share information.

The answer to the question “Where can it be used?” depends on you. The software can be applied to a wide range of subjects, from high-school to postgraduate level. But the degree to which it can be successfully adapted to new situations depends on the amount of time and skill you have:

- Almost anyone can set up a FlyingFish server and use an existing problem set. The instructions in this document will lead you through the process even if you are a new computer user. If you are working only at this level you can have something running with only an afternoon’s work.
- If you are willing to “get your hands dirty”, you can easily adapt existing problem sets by removing problems or changing the order. This will allow you to create custom sets that more closely match your needs. In order to do this you have to be comfortable using the file system on your chosen computer platform. Expect to spend an afternoon and several weekends getting everything right the first time. After that you will be an expert and able to make changes with confidence.
- If you need to augment existing problem sets by creating new problems of an existing type, you will have to do some programming work. Depending on the problem type, this programming could be in Java™, Mathematica®, or a script language. This manual provides information about the libraries and conventions involved, but you will probably need to attend an Advance Education training workshop to get started. Our experience has been that each new problem takes several hours to write and will usually contain “bugs” that are found over subsequent months as the students use it.
- If you need a new *type* of problem then you will have to get help from Advance Education technical staff. The FlyingFish system is extremely flexible but new problem types may require minor extensions to the core of the server. You should only try to do this if you are committed to producing something completely novel, you are an expert programmer, and you have plenty of time and money.

1.4 A note about Web browsers

The issue of browser compatibility is important to us and we try to keep all options open at all times. At the time of writing Netscape Navigator™ v4.5 or later works well on both Windows™ and Macintosh™ platforms. Microsoft™ Internet Explorer® v4.0 has been found to be less reliable when used with FlyingFish, although it can still be used. For the time being we encourage the use of Navigator.

For the most up-to-date information about browser compatibility, and more detail about known problems, visit the Advance Education web site.

2. Setting up FlyingFish

The FlyingFish Environment consists of two parts. There is a platform-specific *server* which can take many forms:

- A library of scripts;
- One or more platform-specific executables (applications);
- A set of Java™ servlets.

These different server forms have arisen historically and not all of them will be supported in the future.

Your main task, however, is to create the other part of the Environment: the problem sets and user databases for your course. These are *completely platform-independent*. This means that, if you invest effort in creating new problems or new problem sets, *they will continue to work properly even if the server changes*. If you need to move your entire course (including all users) to a different computer, or even a different computer platform, the users should not even notice the difference.

You can think of the FlyingFish Environment in the same way that you think about the Web. The Web consists of Web Servers, available for all the main computer platforms, which serve Web files such as HTML and GIF files. The file formats are platform-independent: you can move your Web site to a different server, or change Web server binaries, and the files continue to look the same as far as the user is concerned. The key to this portability is adherence to published standards and conventions. As long as all the Web servers and Web browsers speak the same language, it all works smoothly.

The Web's extraordinary growth is at least partly the result of an inherent simplicity. In its original form it was meant only as a mechanism for publishing documents. The file format for HTML is very simple and can be edited with an ordinary text editor by just about anyone. We have tried hard to have this kind of simplicity in the FlyingFish environment. This is important because, just as we expect the Web and HTML to exist for a long time, we hope that courses written for the FlyingFish Environment will be useful to students for a long time. If we relied on proprietary file formats, then one day the editors for these formats would "go out of date" and your effort in creating the files would be wasted because you could no longer change them to keep them current. These simple, platform-independent conventions are explained in a later chapter.

2.1 Setting up FlyingFish under Windows 95, 98 or NT™

You should read this section if you need to set up a new FlyingFish server on a Windows™ computer. For brevity we will refer to Windows 95, 98 and NT using the generic name "Win32". This name is used by programmers to distinguish the modern 32-bit Windows from the older 16-bit version.

2.1.1 RECOMMENDED SERVER CONFIGURATION

For smooth operation, try to match or exceed all of the recommended server specifications since they are tested by time and by thousands of students.

Hardware:	Pentium 266 with 64Mb RAM and 2Gb HD.
	10 Mbit/s EtherNet TCP/IP connection with a fixed IP address.

Software: Windows 95 or 98, or best of all,
Windows NT™ 4.0 build 1381, with service pack 3 or later.

The server computer may not be busy all the time and (depending on your class size and other factors) you may find that you can use it as usual. In other words, it's possible that the server can be *your own desktop computer*. However, if you find that the computer seems to slow down while students are using it, or if the Performance indicators show that the CPU is very busy, you will have to leave it alone to do its job of serving.

The whole point of a server is to serve files to someone who wants them. So the server will need a decent internet connection. For development work it is OK to have no internet connection or a very poor one (e.g. a modem line only). But for high-volume serving you have to have a fast, stable TCP/IP connection. Your server's IP number is a unique 4-digit number (e.g. 130.95.52.4) which identifies your computer and can be used in the Location field of a web browser. So it is also very important that a high-volume server has a stable IP address. Students would become very frustrated if they could log in using a certain address one day but not the next! If you are in doubt about the type of TCP/IP connection you have, or whether your IP address is stable[3], contact your network administrator.

2.1.2 GETTING READY TO INSTALL FLYINGFISH

The FlyingFish CD-ROM has compressed files on it and you will need a zip utility to decompress them. A public-domain zip utility called PKZip is included on the CD-ROM. If you do not have a zip utility already, you should open the PKZip directory and follow the installation instructions for PKZip.

By default the Windows™ Explorer hides file extensions i.e. a file called thing.ini will appear on the screen as simply "thing". Actually, depending on the settings, thing.ini it might not appear at all! I find this annoying so I usually set the Explorer View Options so that all files are shown, and all file extensions are shown. To do this, select View/Options from the Explorer menu. Set your options to look like the example on the right ⇒

2.1.2 INSTALLING FLYINGFISH

Step 1 Insert the FlyingFish CD. Open the directory /win/ and find the latest version of the FlyingFish package. This will be the file called

---

3 I have observed that some universities use dynamic IP address allocation (e.g. DHCP). Thankfully these 'dynamic' allocation schemes are not always truly dynamic i.e. it is likely that your computer is always given the *same* address! Appendix 3 explains some techniques for determining your server's address.

"FFishXXX.zip", where XXX is the highest integer you can see. At the time of writing the latest version was FFish151.zip.

Step 2 Double-click the FFishXXX.zip file and extract to your C: drive (it is also OK to extract to some other drive – everything will still work).

Step 3 When the extraction is complete you should see a single new directory called C:\FlyingFish.

Step 4 Your FlyingFish server could now run. However it is worth taking a moment to edit the ini file since you will have to do this sooner or later. Edit the file FlyingFish/jellyfish.ini[4] using a text editor like NotePad[5]. You will see something like this:

```
tree/u=../user
tree/t=../task
tree/y=../2000
farm/address=130.95.16.27
farm/port=5555
http/port=8080
admin/email=nscott@mech.uwa.edu.au
admin/name=Dr Nathan Scott
timeout/idle=1800
GMT Offset=-28800
persistent/mozilla=1
persistent/msie=1
license=1anz-a8rq-9y4b
license/expires=31 Dec 2001 11:59PM
license/hostIP=130.95.52.241
license/software=FlyingFish
```

Each line of this file is significant and will be explained later. For now, edit only the following lines:

- Change admin/email to show your own email address;
- Change admin/name to your own name;

---

4 The Windows Explorer has some user-settable preferences. One of these preferences is that certain files are hidden by default. In other words, it's possible that, on your computer, you will not initially be able to see the file "jellyfish.ini" because it is treated as one of these special hidden files. To see the file you will have to alter the preferences of the Explorer. To do that, use the View menu and select the menu item Options. Choose the setting "Show all files".

5 I keep a "shortcut" for NotePad on my desktop at all times. To create a chortcut, right-click the Start menu and select Open. Find NotePad in the Programs directory. Then right-drag it onto the desktop. One of the options you will see is "Create shortcut here". Then, to edit a file using NotePad, just drag the file onto the shortcut you made – easy.

- If you know what your GMT (UTC) offset is, edit the line GMT Offset=...; the value is in seconds.

Unless you know what you are doing it is probably best to leave the http/port line alone for now.

Step 5 Now you should start your FlyingFish server as described below under *Check that the FlyingFish is working*

2.1.3 REMOVING FLYINGFISH

The FlyingFish installation does not alter your computer's guts at all. It does not change the Registry and it does not mess with your menus. So it is easy to remove it: just drag the entire FlyingFish directory to the Recycle bin.

2.2 Check that the FlyingFish is working

In the following examples it will be assumed that you installed FlyingFish onto your c: drive. If you put it on some other drive, you will have to substitute the letter of that drive wherever you see c:.

The FlyingFish does not start running automatically – you have to tell it to start running. Use a Windows Explorer™ to view the directory

c:\FlyingFish\htdocs\

Double-click the file

FFish151.exe[6]

If all is well you will almost immediately see a new window that looks like Figure 1.

Figure 1 Initial appearance of the FlyingFish window.

You will see that there is a warning about your license in bright red type. This warning will go away when you obtain a valid license key and install it in your

---

6 It is possible that the CD-ROM will have a later version of the executable so it may well be called something else like FFish152.exe. It is also possible that your computer is set up so that it hides the ".exe" suffix, and in this case you would see only a file called (e.g.) FFish152.

jellyfish.ini file. Do not worry about this for the time being: your server will work fine (for two hours) and you can always restart it to get another two hours of operation.

The initial window of the FlyingFish server is simple by design; the idea is to get the server running as quickly as possible. There is another display mode in which you can get an overview of what the users of the system have been doing. Select "Show progress" from the "View" menu. You should see something like Figure 2.

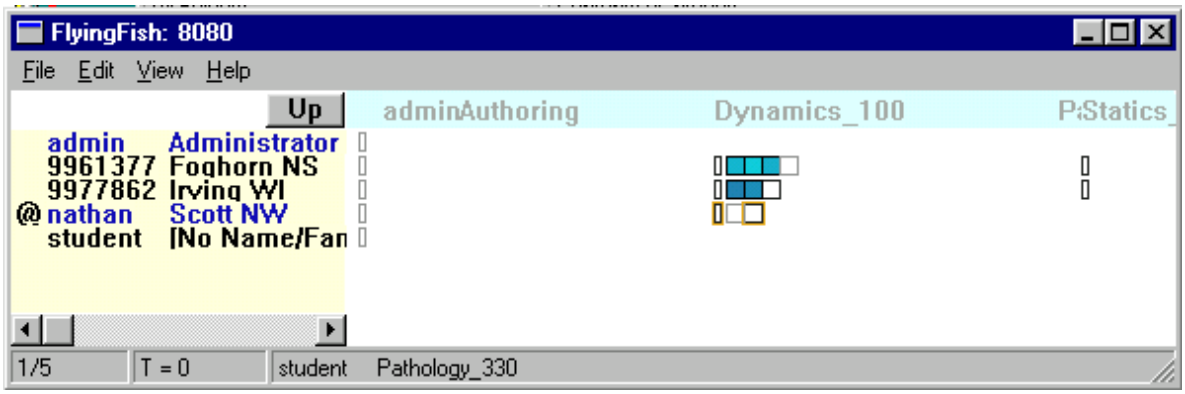


FIGURE 2 FlyingFish progress display.

Before we go on, I should say something about the FishMinder. When you started FlyingFish, you may have noticed that *two* programs came to life: one called FlyingFish and one called FishMinder. FishMinder has a window like Figure 3.

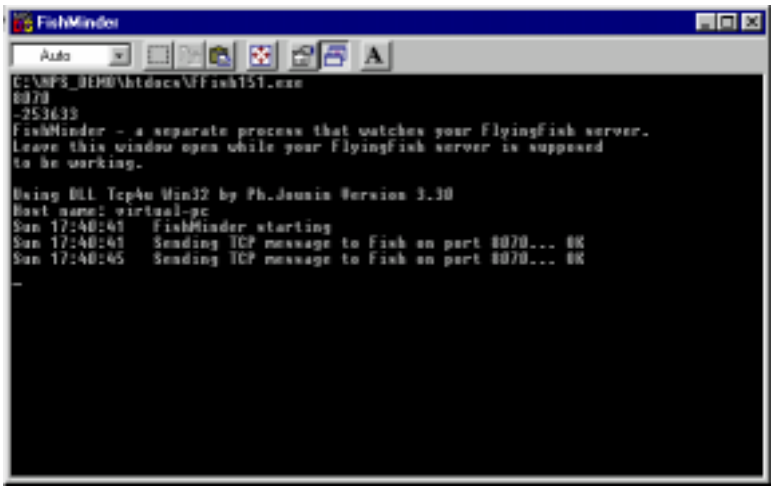


FIGURE 3 The FishMinder window.

Do not be alarmed by the FishMinder – it is harmless and will be explained later on. Essentially it is just a simple process that continually watches your FlyingFish server. In the (unlikely) event that your FlyingFish server should crash, the FishMinder can bring it back to life again.

Now start an instance of your preferred web browser (e.g. Netscape Navigator v. 4.5). Type the IP address of your computer into the Location field, followed by a colon and the port number of your FlyingFish (8080 by default). If you can't see anywhere to type the URL then you need to change the settings on your browser.

For example, you should type something like:

http://130.95.52.48:8080/

If you don't know how to find your own IP address[7], see Appendix 3.

Press return.

You should see a web page like Figure 4.

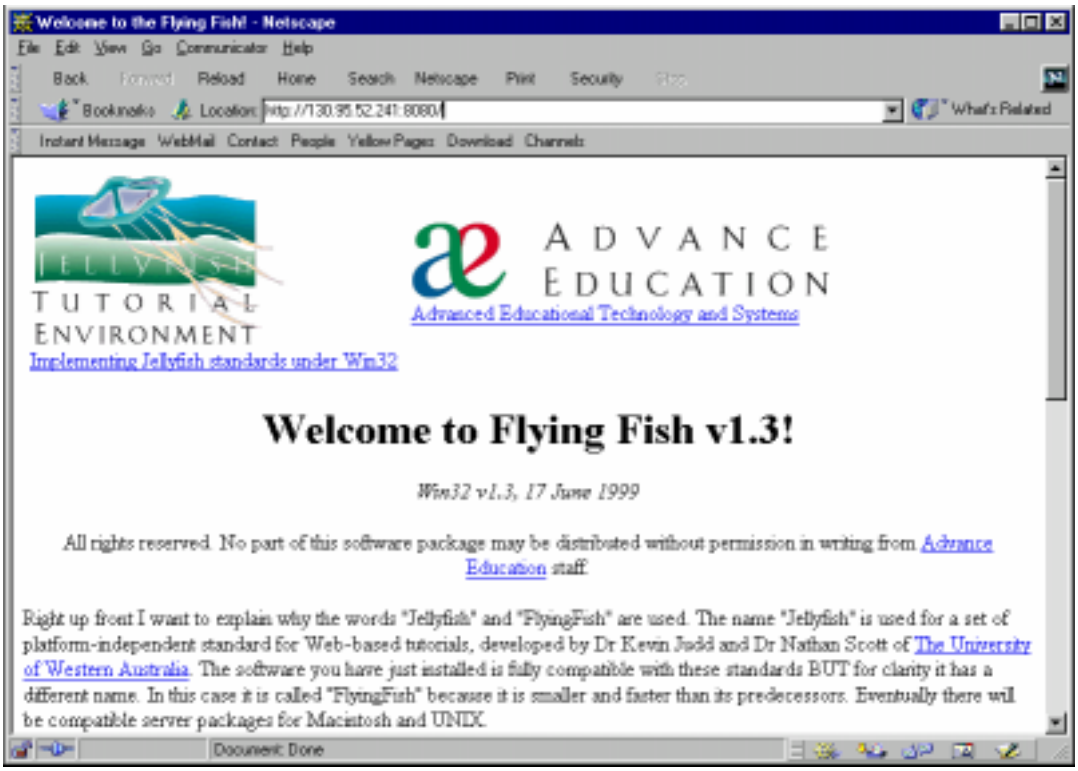


FIGURE 4 Main index.html page.

The main index.html page is the file c:\FlyingFish\htdocs\index.html. Scroll down the page and follow the link to the Login page (Figure 3).

7 Here's a quick way to avoid looking at the appendix. Type http://localhost:8080/

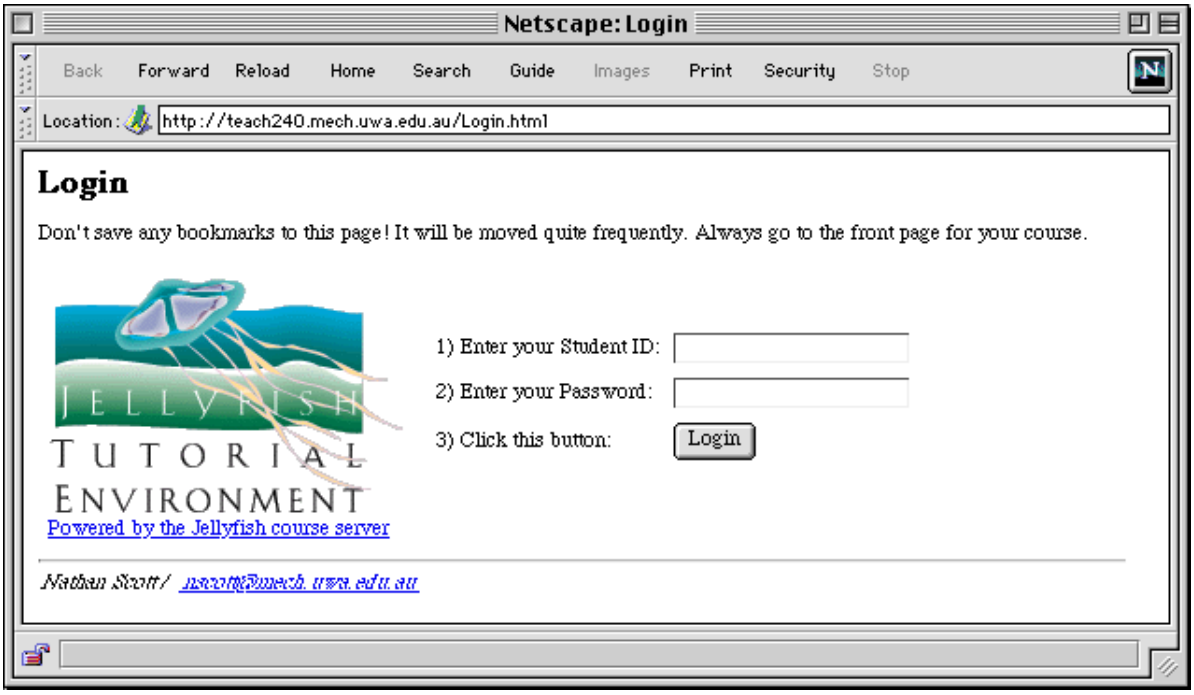


FIGURE 5 The login page.

Type user ID “admin” but leave the password blank. Press the “Login” button. If all is well, you will immediately see something like this:

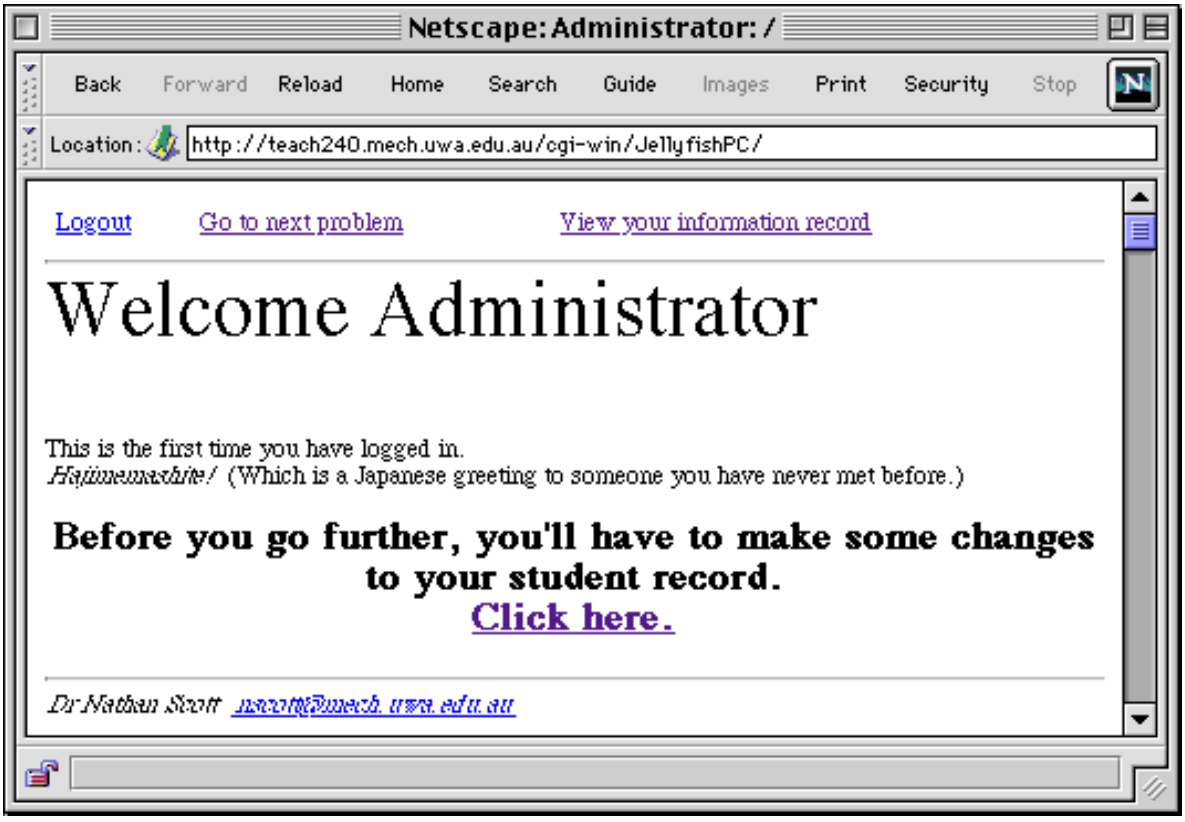


FIGURE 6 The basic front page.

Follow the link marked Click Here and you will see the “information record” page. Students are automatically directed to this page if FlyingFish notices

anything odd about their records, for example if the password is missing or is too short.

FIGURE 7 Setting your preferred name and password.

Enter a new preferred name and password (enter the password twice) and click the Submit button. If your preferred name and password are accepted then the error message at the top of the page will disappear. Follow the link to the Front Page. You will see something like this:

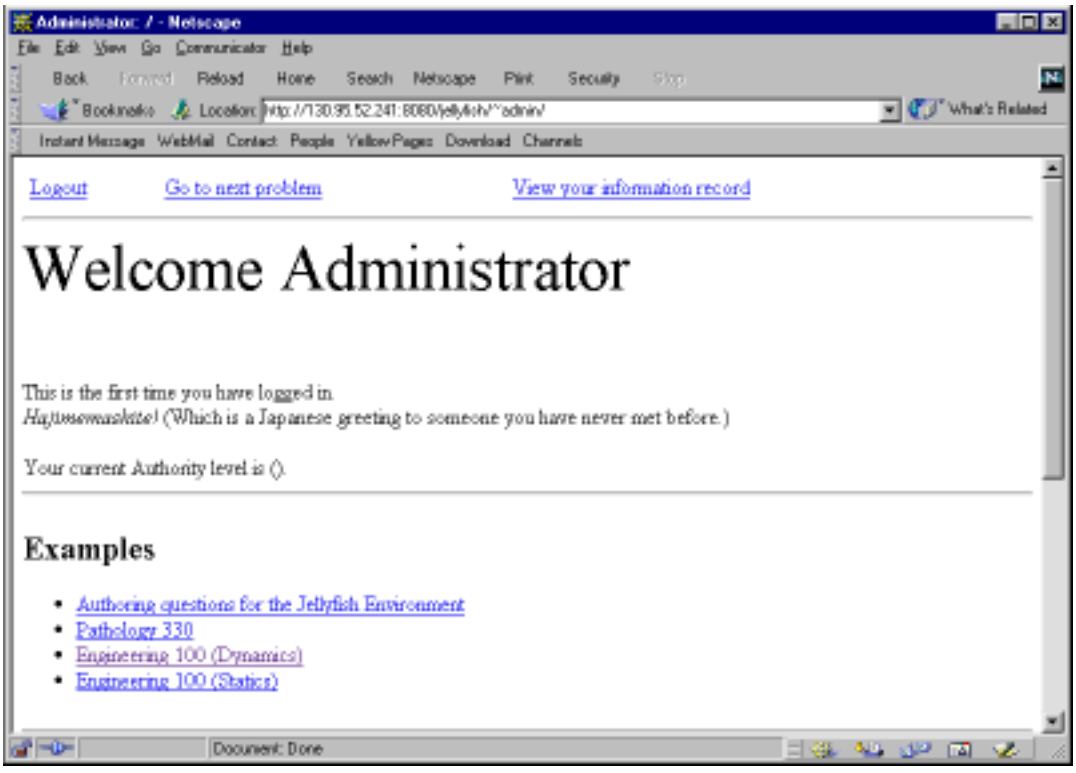


FIGURE 8 The front page after setting a password.

You should explore the links from the front page. The basic FlyingFish package contains only essential files and some examples in the Authoring area. To look

at the Pathology, Statics or Dynamics course material, you will first have to install some more files.

2.3 Installing a course from the CD-ROM

It's easy to install a course from the CD-ROM.

1 Quit the FlyingFish if it is running.

2 Open the *courses* directory on the CD

3 You will have to unzip the files *into* the correct final directory. Do not unzip them onto your hard disk because you might then have hundreds of unwanted files scattered everywhere. For example, if you are installing the Dynamics_100 problem set you would unzip it into the directory

```
c:\FlyingFish\task\Dynamics_100\
```

4 Restart the FlyingFish.

The basic FlyingFish installation has "dummy" directories in the task tree so that you can see where to install course material – feel free to replace them.

3. Flying with the Fish – examples of what this thing does

In this chapter it is assumed that you have done the basic installation of chapter 2, without necessarily installing any additional course material. You should start the FlyingFish server and then log in as described above.

3.1 Variables

If you logged in as "admin" you'll notice that on the front page there is a large welcome message for someone called "Administrator". Later on we will create additional user records and you will have your own private identity and name. The point here is that the FlyingFish server has returned a page which is customised *for the person who has logged in*. It can do this because it has records about each user. The way in which the records are inserted into the Web page you see is very simple and elegant, and it will be explained in a later section.

If this is the first time you have logged in you will see a message saying so. This message will not appear the second or subsequent times you log in. The message is actually enclosed in a *conditional statement* that is testing the value of a user variable. The syntax for this is explained later. Remember, this is a quick tour to get you started – details later.

3.2 Java applet questions

If you follow the link from the front page to Authoring questions for the FlyingFish Environment and then Question applets you should arrive at a page with a Java applet on it. You should see the message "Starting Java" at the bottom of your browser window, and eventually you should see something like this on the screen:

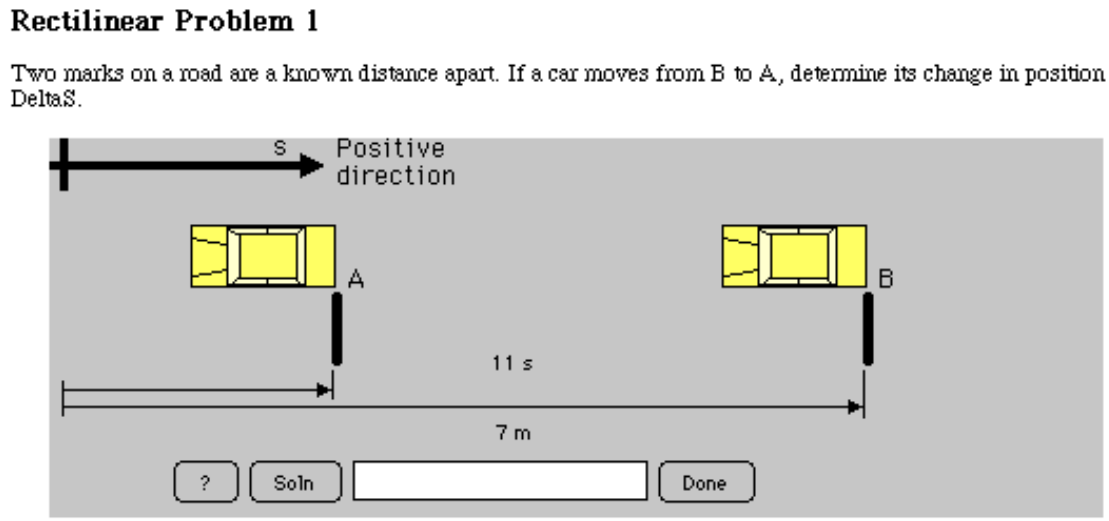


Figure 9 A typical numerical answer applet (Authoring/JavaQuestionsI/)

Follow the instructions on the page and see how the applet responds. If you can, finish the problem by getting it right and then follow the link "Go to next problem" (at the top of the page).

3.3 Multiple choice question (MCQ/)

It is my view that multiple-choice questions are boring for students and have serious educational shortcomings. However it seems that, as far as many

teachers are concerned, they are the height of computer-based training. So for completeness we have included an example. Follow the link from the Authoring page to Form-based Multiple-Choice Questions.

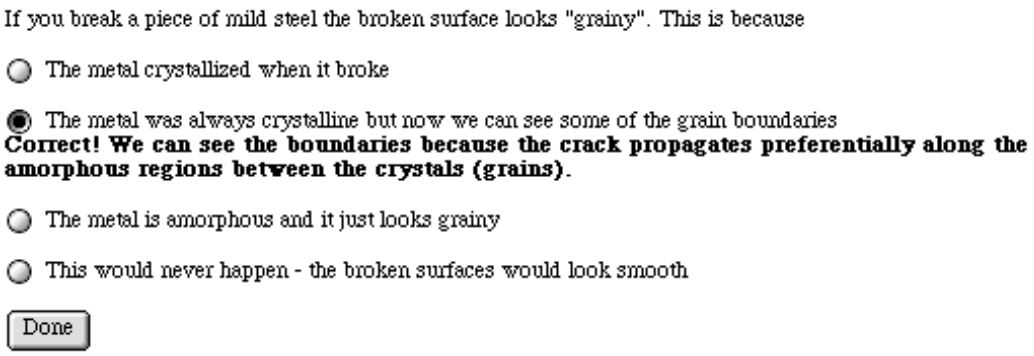


Figure 10 A simple Multiple-Choice Question (Authoring/MCQForm/)

3.4 Drag-and-drop doughnut Java questions

From the Authoring page, follow the links to Drag-and-drop Java Micrograph Questions and then Breast Oncology 2. You should see something like this:

FIGURE 11 A Drag-and-drop doughnut question.

If the applet does not appear then it's possible that your computer has run out of free RAM while trying to load the large images in the problem, or that Java is not enabled.

Figure 9 shows an example of a 'drag and drop doughnut' applet. This question type has an image and some 'markers' which the user must place on the image. Initially all the markers are along the right-hand edge of the applet. Try to "drag" one of the markers onto the image using the mouse, and then click the Done button. A small tick or cross will appear next to the marker you dragged. Now try holding the mouse cursor over the marker without clicking. You will see a short message explaining why your attempt was incorrect (if it was).

This type of problem is very versatile and easy to set up – more later.

3.5 Insert-the-word questions

This problem type presents a paragraph of text with some words omitted. The student is asked to select words from pop-up menus to fill the gaps.

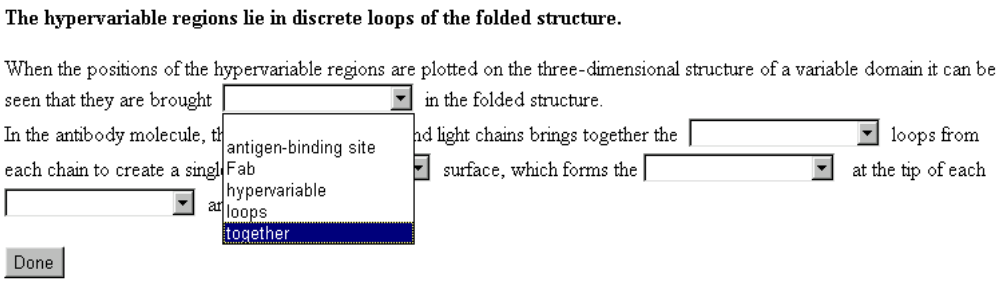


FIGURE 12 An insert-the-word (ITW) problem

The Done button can be pressed at any time and the student's current work is checked. If any word is correctly inserted, the pop-up menu is removed and a bold coloured word replaces it.

3.6 Tutor marked

Sometimes there is no substitute for a written answer to a problem. Such problems must be assessed by human tutors because computers are very bad at understanding free text (and I am not smart enough to even try this!). It is easy to set up an HTML form which students fill in with their answer. Staff can then read the work and give marks.

3.7 Mix and match

The problem types shown in this section are representative only: if you are willing to do some development work you might be able to create a completely new type of problem for your students. The Java question type is particularly flexible since *anything you are able to program* can in principle be made into an assessed task.

Note that the problem types shown can be mixed up in any order you like. Each problem type obeys certain minimum standards to keep the system informed about what the student has done. So if you want a problem set that has eight Java style questions followed by several insert-the-word style, that is OK. The detail of how to set this up is explained in a later section.

3.8 On-line editing of HTML

If you are logged in as an administrator you should see a small link at the bottom of each web page. The link says "Edit this file". If you click on the link you will see an HTML form containing text which you can edit. To submit changes just press the Submit button. Note that the edit function is nothing to do with HTML editors that are built into your web browser.

You can also see a directory of files and directories wherever you are by using the small "Directory" link at the bottom of each page. This release supports a basic file-upload tool:

Future versions of the system will support more advanced file management features such as file deletion and duplication. For now, these tasks must be done at the server computer using the Windows™ Explorer®.

4. Understanding the file structures

4.1 Essential background: default files and relative references

When you are using your web browser, you are continually making requests of remote servers. At all times your web browser keeps a URL string that uniquely describes the web page or picture you have asked for (the Location). If you look at a web page containing many graphics, sounds, movies, Java applets and so on, your browser must actually make a separate request for each item on the page.

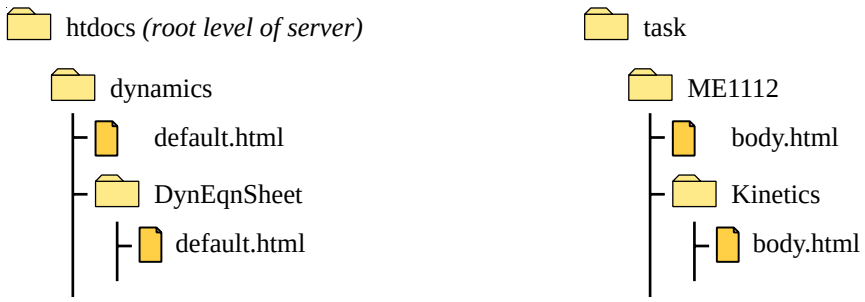
Consider the following URLs:

```
http://www.mech.uwa.edu.au/dynamics/index.html
http://www.mech.uwa.edu.au/dynamics/
```

The first URL is a request for a file called /dynamics/index.html which is (hopefully) on the server called www.mech.uwa.edu.au. But the second URL does not terminate with a file name. The web server www.mech.uwa.edu.au will understand the second URL to mean "please return the *default file* at the location /dynamics/". The default file name is usually index.html but some web servers define a different name e.g. default.html or welcome.html.

It may seem a little odd to refer to a file without giving the exact name, but it has some advantages. It makes the URL shorter and it also give a directory on a hard disk an identity of its own, a sort of "entry point".

The Web defines a very useful behaviour called *relative referencing*. This means that a given document (e.g. index.html) can contain a reference to another document which is expressed in a *relative* way. Refer to Figure 11.



(a) Ordinary web site (b) Jellyfish task tree

FIGURE 13 Directory trees and relative references

In an ordinary Web site (Figure 11a) the file dynamics/default.html could contain the following HTML fragment:

```
<a href="DynEqnSheet/">Standard Equation Sheet</a>
```

If the user clicks on this Anchor the browser understands that the document

```
(root level)/dynamics/DynEqnSheet/
```

is desired.

Of course the Web server knows that the document

```
(root level)/dynamics/DynEqnSheet/default.html
```

is to be returned.

The file reference “DynEqnSheet/” is very simple and has a useful property: if the directory “dynamics”, with all its contents, is moved to some other part of the directory tree, the *relative reference* “DynEqnSheet/” in the document dynamics/default.html remains valid. It is as though the directory is “pointing” to something inside itself.

The process of “surfing” a given web site, that is, looking around at the various files in the site, can thus be seen as either adding new elements to a path (to go deeper into the directory tree), or removing them (to climb out of a directory).

Note that the convention is that a *relative* reference has no leading “/” character. An *absolute* reference, however, begins with a “/”. This convention will make more sense when we come to discuss the insertion of variables in the FlyingFish Environment. Also note that it is possible to make relative references that refer to directories or files at a higher level in the directory structure. To do this you use the “climb up” operator “../” e.g.

```
../../directory1/file.html
```

climbs up two directory levels from the current location, and then goes down into the directory called directory1. If there is a file called file.html, it is returned.

4.2 Special directories

The FlyingFish installation on your hard disk consists of a directory with a number of sub-directories and many files. Some of the directories below the FlyingFish directory are special because

- they are named in the FlyingFish/htdocs/jellyfish.ini file so the FlyingFish knows about them; and
- they contain specific kinds of files and sub-directories.

See Figure 11.

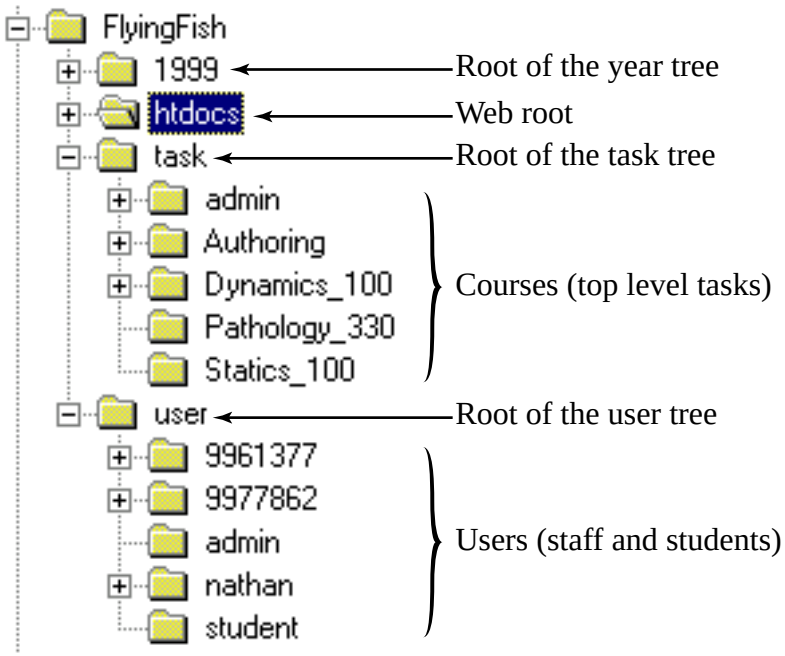


FIGURE 14 Special directories in the FlyingFish tutorial environment

The word “tree” is used to describe the directories because they have a tree-like structure. A directory contains sub-directories and files, and the sub-directories can contain sub-sub-directories and so on – like the branches of a tree that divide repeatedly until they end in leaves. This tree-like property of the file system is heavily exploited in the FlyingFish system – more later.

4.3 The web root and basic web server function

The FlyingFish server is mainly intended to serve course material i.e. web pages containing problems for students to solve. Such course material web pages are typically created “on the fly” by the server i.e. they are created as they are needed based on the contents of various files. There are times, though, when nothing so complicated is required. This is why there is an “ordinary web” root in the system.

If you start your FlyingFish server as described in chapter 2, you will notice that the address of the login page was something like

http://130.95.52.48:8080/login.html

This is a request for the file login.html that is in the htdocs directory. The default file name is index.html.

The FlyingFish web server is deliberately small and simple and as a result it is very fast. It supports the original HTTP1.0 specification only and does not support CGI. If you only want to serve normal HTML web pages, images, movies and so on, the FlyingFish web server should serve your needs. However if you want more elaborate behaviour you should probably also install an additional dedicated Web server such as O’Reilly WebSite (http://www.oreilly.com/). Note that the two servers (FlyingFish and your chosen Web server) can exist side-by-side on the same machine provided that each one uses a different port number.

4.4 The task tree and the current Branch

Consider the task “get a degree”. This top-level task is achieved by working on a number of sub-tasks – the semesters of the degree program. Each semester there are usually a number of courses to be done – subtasks of the semester – and each course usually has a number of assignments and exams – sub-sub-tasks. This tree-like structure goes down to a very fine level of detail: the finest grain of tasks could be sub-parts of a single tutorial problem.

In designing the FlyingFish system we have tried to use straightforward conventions. So it was natural to exploit the tree structure of the file system (directory tree) to express the tree structure of the tasks and sub-tasks. The task tree root can be seen as the highest task known to the system; the sub-directories of the task directory are sub-tasks and so on. A task can be anything from a whole degree program to a single tutorial question – the system treats the tasks in exactly the same way.

Because the tasks form a tree, it is possible to specify a unique path to a particular task, just as you might specify a path to a web-resource in a URL. The path to a given task is of course relative to the root of all tasks, the task tree. So let us suppose you want to look at the web page for a task called Authoring/JavaQuestionsI/. You might initially think that the URL would look like this:

http://130.95.52.48:8080/Authoring/JavaQuestionsI/

However this would not work because the FlyingFish server will interpret this as a request for an ordinary web page; it will try to find the file

FlyingFish/htdocs/Authoring/JavaQuestionsI/index.html

Which is not at all what you were hoping to see and probably does not even exist.

To distinguish requests for task pages from requests for ordinary web pages, an addition to the URL is needed. See Figure 15.

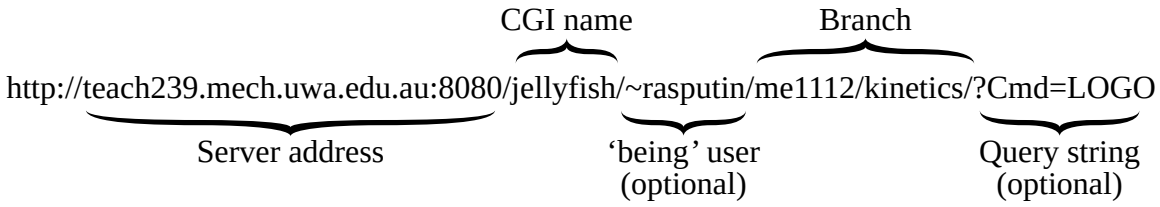


FIGURE 15 Parts of a URL.

The FlyingFish server sees the keyword “jellyfish” after the web server address so it knows that you want to see something generated “on-the-fly” from files in the task tree rather than some ordinary web page. The server parses the remainder of the request to pick out further detail such as the path to the task and something called the query string (more about this later).

The path to a particular task has a special name: it is called the *Branch*.

The FlyingFish server has its own default-file convention. Confusingly, the default file for task-tree-web-pages is called body.html[8] rather than index.html. The FlyingFish will thus interpret the URL of Figure 12 as a request for a document at

FlyingFish/task/me1112/kinetics/body.html

But the FlyingFish server does not simply return a copy of the body.html document, as an ordinary Web server might. The document is *processed* before it is sent to the user. The processing which is done is one of FlyingFish's most powerful features and is explained in a later chapter.

Relative references are possible – indeed, highly advisable – in the task tree as well as in the ordinary web tree. If the document

FlyingFish/task/me1112/kinetics/body.html

contains an anchor

<A HREF="down/">Go down</A>

Then clicking on the anchor will point the browser at

FlyingFish/task/me1112/kinetics/down/body.html

note that the trailing "/" character in the relative reference "down/" really is significant. If we were to leave the / off and make the anchor

<A HREF="down">Go down</A>,

then clicking on the anchor would point the browser at an address

FlyingFish/task/me1112/kinetics/down

All would seem well enough because the FlyingFish would return the correct body.html document, however things could go seriously wrong after this point. The problem is that the browser literally "tacks on" additional Branch elements as you go down into the task tree, and it literally "strips" them as you go "up". It is not smart enough to know which branch elements are files and which are directories. So if you tried to go down again by extending the branch with "below/", you would effectively be pointing the browser at

FlyingFish/task/me1112/kinetics/below/body.html

– in other words, we will have lost the path element "down" because the browser erroneously treated it as a file name.

If this discussion has given you a headache, don't worry about it. Just remember to put a trailing slash character on the relative path to a directory. It does no harm to cultivate the habit of reading "thing/" as "thing directory".

4.5 The year tree and parallel directory structures

The year tree is used to store information that changes from semester to semester, such as student visit counts, forums and deadlines for assessed problems. This sort of volatile information is not portable: if you were to give a task directory (e.g. a whole course) to someone, they are unlikely to be interested in your deadlines.

---

8 The standard installation starts with "body.html" as the default file name everywhere. However it is possible to change this and even to have different default file names in different parts of the task tree. This is explained in the section on the mapping file.

So how does the FlyingFish associate deadline information with task information? By the use of the Branch. The Branch can be “tacked on” to any of the root directories. Consider the Branch me1112/kinetics/.

TABLE 1 Using the Branch to identify tasks, deadlines and student records

Tree	*Root directory*	*Full path after addition of branch*
task	FlyingFish/task/	FlyingFish/task/me1112/kinetics/
year	FlyingFish/1999/	FlyingFish/1999/me1112/kinetics/
user	FlyingFish/user/	FlyingFish/user/[user ID]/me1112/kinetics/

The use of the user ID in the last case will be explained below.

The important new idea here is that there are *parallel directory structures* in the system. The Branch is more than just a unique identifier for the current task: it can also be used as a sort of key when getting information about other things (such as the progress of a user with respect to the task in question).

All the information that can be accessed using a particular Branch string is related, even though it is scattered widely in the computer file system. All the information available at a particular Branch can be seen as belonging to a virtual directory which we call a *node*. Just to make things interesting, in this manual the word node is also sometimes substituted for “directory”. The word node is drawn from computer science nomenclature for one element of an abstract tree structure.

4.6 The user tree

Each user of the system appears as a subdirectory of the user root (refer to Figure 11 above). To add a user one must create a fresh user directory and put certain files in it (containing the name and other details about the user), a process that is explained in a later section. The user directory also stores all information about the work a user has done. As a user completes tasks, the FlyingFish server adds subdirectories and files to the user directory *in parallel* with the directory structure of the task tree. See Figure 16.

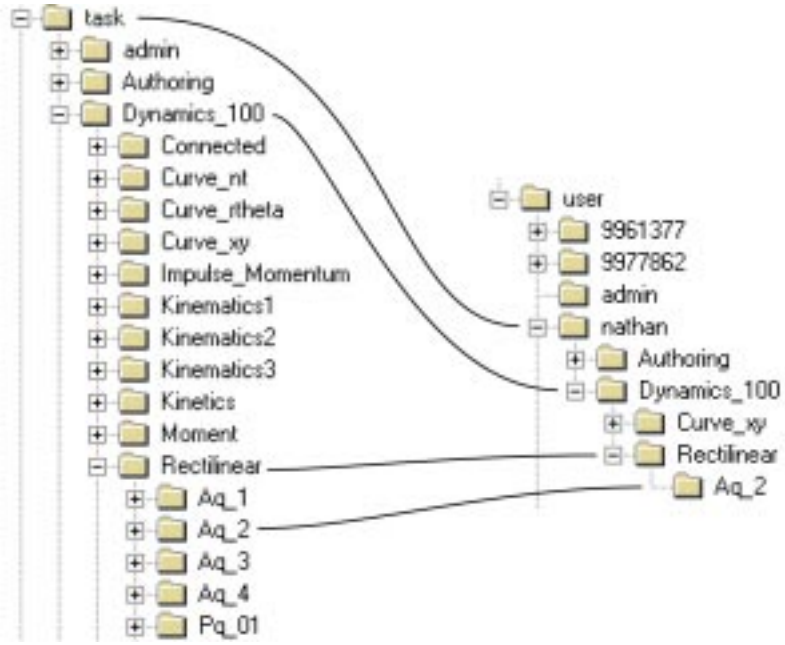


FIGURE 16 Student work stored using a parallel directory structure

As an example, suppose that we wish to check whether the user with ID "nathan" has completed the task at Branch Dynamics_100/Rectilinear/AQ_2. If the user has completed the task, the FlyingFish will have created a file at

FlyingFish/user/nathan/Dynamics_100/Rectilinear/AQ_2/

The file will be called Done and it will contain certain information about the user's work (more about this later).

4.7 The UDB file format and the extended path

The FlyingFish tutorial system consists of a great many files and directories. Some of the file formats are already familiar to you: HTML, GIF, JPG, java class files and so on. The environment defines only one new kind of file, called a Universal Database File (UDB). These UDB files are plain text with a small amount of additional structure. In other words, you can read them and edit them with a simple text editor like the Notepad.

Here is an example UDB file from the path

FlyingFish/user/nathan/user

```
Authority=Developer
Course/2A1=1
Course/M155=1
Course/SM155=1
Name=Nathan Scott
Name/Family=Scott
Name/Given=Nathan
Name/Initials=N
Name/Preferred=Nathan
Password=mudflap
UserID=nathan
```

Note that

- The file consists of key=value pairs e.g. Authority=Developer;
- If a key contains the / character, a sub-key is defined and these key/subkey combinations describe a tree structure;
- Line breaks are significant and each key=value pair appears on a single line.

The UDB file format has some nice properties:

- It is very flexible – you can add additional key=value lines without affecting the meaning of the existing lines. In other words, the *order* of key=value pairs in the file is *not* significant;
- Line breaks can be CR, LF or CRLF pairs;
- It is plain text so it is completely cross-platform;
- It is simple and human-readable which is a great advantage if you have to make manual changes.
- There is nothing secret or proprietary about the file format so it is not likely to go out of date.

There is one additional property of the UDB file format that may not be immediately obvious. Suppose that we want to access the Family Name of this user. We can do this with an *extended* URL which not only identifies the UDB file, it goes even deeper and identifies a *single key within the file*. The path

FlyingFish/user/nathan/user/Name/Family

identifies the value “Scott”.

The usefulness of the extended-path idea will become clearer later on when we explore the syntax for substituting variables into HTML documents.

4.8 File caching vs. file editing

The FlyingFish is optimised for high-volume student use rather than for the relatively infrequent editing work of staff. Part of the speed optimisation is a series of internal caches for frequently accessed files.

The disadvantage of these caches is that they may persist even when you have modified a file on the hard disk. See Figure 17.

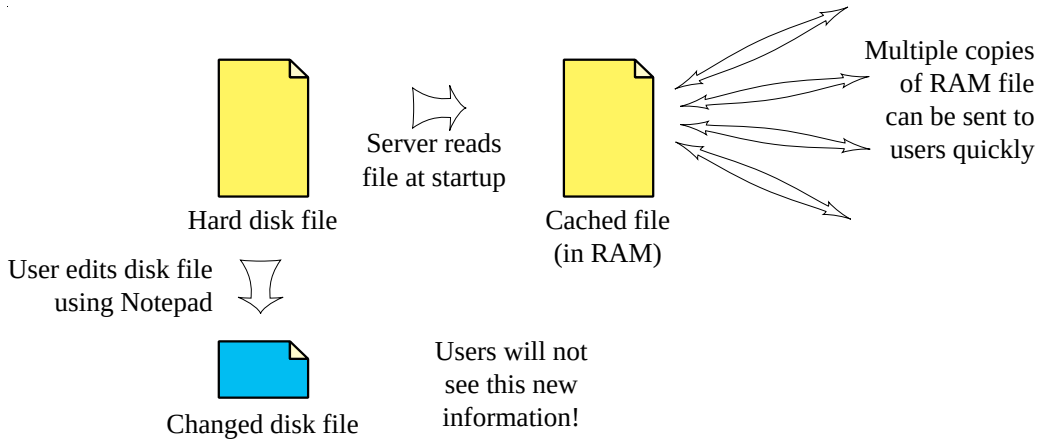


FIGURE 17 The problem with caches

So to successfully modify served files (i.e. the contents of the task, year and user trees) you must take one of two approaches:

1 Quit the FlyingFish server before making changes, then restart it.

2 Use only the on-line administrative and editing tools (more about this later).

4.9 Parallel directory structures are not foolproof

As a user surfs the task tree, new directories and files are created in his or her user directory. The new directory structure is an exact analogy of the structure in the task tree. This is how FlyingFish knows which tasks this user has done, and what marks were awarded and so on. Be aware that problems can arise if you make changes to the task tree after some students have started to solve problems. Suddenly the “template” in the task tree will no longer match the records stored in the user directories, and the results can be unpredictable and are usually bad. Future releases of the system will include administrative tools to allow editing of “live” trees (e.g. parallel editing of 300 student records!). For now, it is a good practice to avoid making major changes to the task tree after the start of a semester.

5. Creating and managing user records

5.1 Impersonating another user

The FlyingFish generates most web pages dynamically based on user information. In other words, the page *you* see at a certain Branch might look quite different to another user. So it is helpful to have a way to quickly check the way the pages look to other people. You could do this by getting the UserIDs and passwords of the other users and logging in using these details, but there is another way.

The syntax of the URL to view a particular Branch can include the ID of a user like this:

http://130.95.52.240:8080/jellyfish/~student/Dynamics_100/Kinetics/

The path element ~student is understood to mean “show me this page as if I were the user called student”. It has to have the ~ prefix and it must be the first path element after the jellyfish keyword. By entering another user’s userID here, you create a distinction between the *actual* system user and the “being” system user (the impersonated user).

Low-level system users (e.g. Student users) can enter this additional path element if they wish but it will be ignored and they will only see their own pages.

- The *actual* user is always the user who *entered a password* to authenticate him- or herself to the server. The authority level of the *actual* user is taken seriously by the server and it limits or grants access to executive functions.
- The *being* user is whatever user name appears after the ~ in the URL, provided that the actual user has sufficient authority to impersonate the being user. If the *actual* user has insufficient authority, the being userID is ignored and might as well be the *actual* userID.

A simple tool is provided for quickly getting access to student pages (without having to remember their IDs). If you visit the user information page (admin/) you will see some administrative links at the bottom of the page including a Class List. Follow the link to the Class List and you will see something like this:

If you click on the ID of a user, you will immediately start impersonating him or her. To stop impersonation, edit the Location field to remove the ~user path element.

The impersonation mechanism is very useful for editing student details such as name and password, for awarding marks to students, for correcting marks, and even for creating new user records, as we will see in later sections.

*5.2 Creating a single user record the **hard** way*

This really is the *hard* way and an *easy* way is explained below. This section is only here to show you what a minimal user record contains.

Each FlyingFish user has a single directory in the (FlyingFish root)/user directory on the hard disk. So one way to create new user records is to duplicate an existing user directory[9], and then to modify the contents of the directory. The important, persistent information about a user called XXX is stored in the UDB file

(FlyingFish root)/user/XXX/user

Here's an example of a minimal user UDB:

```
Authority=Developer
Course/2A1=1
Course/M155=1
Course/SM155=1
Name=Nathan Scott
Name/Family=Scott
Name/Given=Nathan
Name/Initials=N
Name/Preferred=Nathan
Password=mudflap
UserID=nathan
```

The important features of the file are:

1 It contains a key for the Authority of the user. Currently supported values are "Student", "Helper", "Controller", "Designer" or "Developer". Don't use any other strings – they won't be recognised. It's probably a good idea to use only "Student" and "Developer" for now.

2 There are any number of Course keys which tell the system which courses the student is supposed to be doing. The Course keys can be anything you like but it is a good idea to make them match the names of the tasks in the task tree.

3 The Name and Password keys are fairly self-explanatory.

4 There must be a UserID key and it must be the same as the name of the directory that the user UDB is in. This is an annoying restriction which will be removed in a future version of this system. For now, just make sure it is true.

5.3 Creating a new user the easy way

A user of sufficient authority is allowed to create new user records. The process is very easy. Recall that it is possible to impersonate another user using an extension of the URL syntax, e.g.

http://130.95.52.240:8080/jellyfish/~rasputin/Dynamics_100/Kinetics/

But what will the server do if you try to impersonate a user that does not exist? If you have sufficient authority, and if your preferences are set correctly, the

---

9 Don't forget – always quit the FlyingFish before working with the files manually!

FlyingFish will create a new user record! Try the following steps to make yourself a user account:

1 Log in as admin (a user with a high level of authority).

2 Follow the link to the information page (admin/). You will see information about the admin user account.

3 Make sure that the check box “Automatically create missing students” is checked. If you had to change the setting, be sure to Submit the form so the change is noted.

4 Edit the browser location field by changing it from

   http://130.95.52.240:8080/jellyfish/~admin/admin/

   to

   http://130.95.52.240:8080/jellyfish/~[yourID]/admin/

   where of course you will substitute your preferred user ID for [yourID].

   Note that user IDs should only contain ordinary alphanumeric characters: avoid the use of punctuation marks and other odd symbols. Never use spaces, the ~ or / symbols, or the question mark (?) – these might be interpreted as path elements and would cause very strange behaviour indeed.

5 If all is well the admin page will change so that most of the fields are blank and you have an opportunity to fill in your details. Note that the Authority string should be one of “Student”, “Helper”, “Controller”, “Designer” or “Developer”.

6 Submit the form.

7 That’s it. Your new user has been created. Go to the Login page and try to log in as your new self.

5.4 Creating many student records at once

A common administrative problem at the start of a new course is to create a user directory for several hundred students at once. FlyingFish provides a crude, but effective, solution to this problem.

First you have to create an input file. The file should be PURE TEXT, i.e. **not** a proprietary format such as Microsoft® Excel™. It should consist of tab-delimited text. The first row of data should be a list of column identifiers. Subsequent rows are student data.

StudentNo	Surname	Initials	FirstName
8431837	Scott	N W	Nathan
9422213	Abe	G	George
...	...	...	...

The order of the columns is not important, provided that the StudentNo column appears first. It is OK to have additional columns of data: the data will be stored with the new student records.

The name of this input file should be the name of the course followed by the string “.txt”. For example, if your course is “Dynamics1”, create a text file called “Dynamics1.txt”.

Choose the FlyingFish menu item “File/Load Class...” and locate your input file. FlyingFish will think about the input for a few seconds and ... ****

You should now see a great many new directories in the user tree. Check that the UDB entries for students seem reasonable.

5.5 The FlyingFish progress display (monitor)

When you first start FlyingFish it shows a picture of a Fish and a few words. However a detailed user display is also available. Choose “Show Progress” from the View menu.

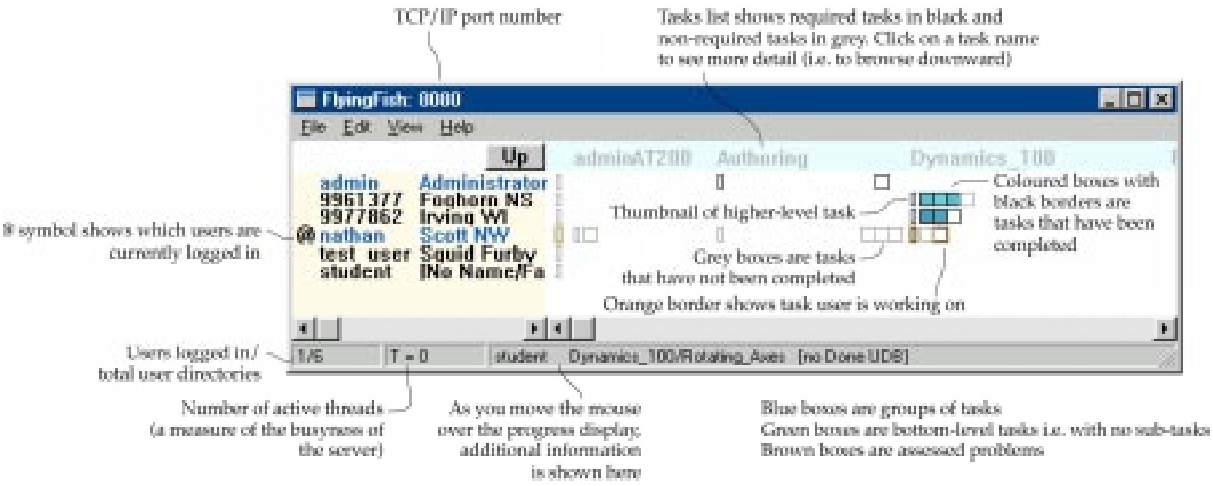


FIGURE 18 A visual display of student progress.

The progress display serves several purposes:

- It allows you to see the progress of your whole class at a glance, which will help you to set appropriate deadlines.
- It allows you to see if any students have fallen behind.
- It is possible to change student marks by simply clicking and dragging.

The progress display would not be much use if it could only show the top level tasks. To see more detail about any task, just click on the name of the task in the pale blue bar at the top of the progress display. If you do not see the name of a task it is possible that it is obscured by the name of some other task. In this case, watch the status bar at the bottom of the window – it always shows the full path name of the task you are navigating to.

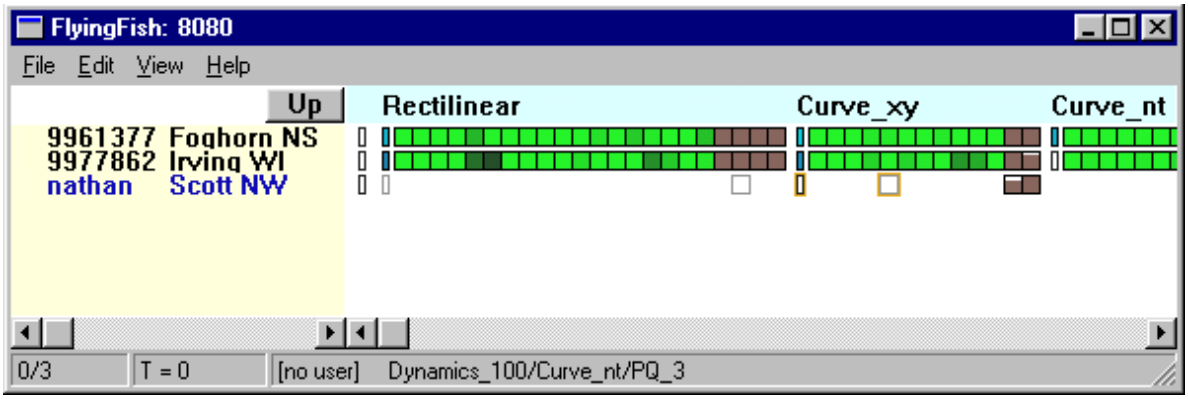


FIGURE 19 More detailed view of the Dynamics_100 course

To return to a higher level view (i.e. to view a shorter Branch), click the Up button (just above the list of users).

To change a student's mark you must first navigate to a view of the task in question that shows the lowest level of detail. You can only change marks on assessed problems. Click on the brown box for an assessed problem and then drag the mouse vertically. You will see a display in the status bar about what the mark *was* and what it *will be* when you release the mouse.

Users can be sorted in various ways – try the View menu to see the current options. If you sort by Progress, students who have fallen behind will appear at the bottom of the list.

I apologise for the slowness of the display; if you are viewing a large class it can take quite a few minutes to just find the students you are interested in. The display is slow because it stores a minimum of information in RAM – most files are left on the hard disk most of the time. This is a deliberate design decision to reduce the chance of filling up available physical RAM while the system is in use.

At the moment there is no way to see the progress display unless you sit in front of the server computer. Future system releases will include a range of other monitoring tools including a Java version that runs on a web page and can be seen from any remote browser.

6. The FlyingFish processor

In Chapter 3 you surfed around a FlyingFish site and saw some of the possible problem types. In this chapter I'll explain the various functions of the FlyingFish document processor that generated the pages you saw.

Consider the following body.html document (task/body.html):

```
<font size=10>Welcome @{status/User/Name}</font><p>

[@{user/Login}<2 ?
This is the first time you have logged in. <br>
<i>Hajimemashite!</i> (Which is a Japanese greeting to someone you have never met before.)<br>
:
You have logged in @{user/Login} times.<br>
]

[@?{user/Login/Prev/Time} ?
Your last login was @{user/Login/Prev/Date} from
@{user/Login/Prev/Host}<br>]<BR>

["@{/status/User/Auth}"!="Student"?Your current Authority level is
<strong>@(u){/status/User/Auth}</strong>.<BR>]

<HR>
<P>Click on the name of your course:</P>
<UL>
<LI><A HREF="MS451/">Mechanical Systems 451</A>
</UL>
```

You can view the document yourself by dragging the file onto a simple text editor such as the Windows™ Notepad. Do not try to use a complex piece of software such as Microsoft Word™, because the latest versions are too clever and they will try to interpret the HTML tags rather than just showing the text.

The file looks very much like standard HTML except that there are some unusual tags. In the following sections the meaning of each of the special tags will be explained.

6.1 Inserting a variable

In simple terms the tag "@{status/User/Name}" means "replace this tag with the user variable status/User/Name". To see exactly how FlyingFish deals with this, use your file system to find the file (FlyingFish root)/user/admin/status. Open this file with the simple text editor. You'll see something like this:

```
Branch=/MS451/Curve_rt_rel/PQ_6/solution/
Branch/Prev=/MS451/Curve_rt_rel/PQ_6/
Branch/Suggested=/finished.html
Code=1651327636
Code/Expires=902398452
Code/Prev=1539142932
Code/Prev/Expires=902398436
```

```
Remote/Addr=130.95.52.239
Request=1587
Tick=903367852
Tick/Prev=888149163
User=admin
User/Auth=Developer
User/Name=Administrator
User/Seed=1752457583
```

This is an example of a UDB or Universal Data Base file. FlyingFish uses files of this type to store most information about users and some information about tasks. The idea is very simple. Each line of the file contains one key=value pair. The key can contain hierarchical divisions, for example User/Name. Each user has a directory and these directories contain a number of files, for example user and status UDBs. Collectively these UDB files describe who the user is and what he or she is allowed to do. Later on we shall also see that each user directory stores an image or analogy of the entire task tree, as a record of what work is said to have been completed.

So we can now interpret the tag @{status/User/Name} fully:

- The @ symbol tells FlyingFish to insert a file or variable;
- There is no leading "/" character before the word "status", so this is a relative reference to something;
- By default variables are assumed to be *user* variables. There is a syntax extension to allow other types of variables to be accessed – more later;
- So FlyingFish knows that it is looking for something with absolute path

  (FlyingFish root)/user/admin/status/User/Name
- The FlyingFish server searches for a file in the absolute path and loads the file (if found). In this case it finds the user/admin/status file. Then it searches the file for the key User/Name and finds the value "Administrator". This value replaces the tag "@{status/User/Name}" and is therefore seen by the user when the processed HTML is received.

6.2 Tree-climbing references

The full syntax for inserting a file or variable is

@(modifiers){path name}

If the modifiers are missing then they are set to the default value (u) – explained below.

Here are the currently defined modifier strings and their meanings:

Modifier string	Meaning
u	Search in the current branch of the User tree
t	Search in the current branch of the Task tree
u^	Start from the current branch of the User tree. Search by climbing upward through the User tree until either the target is found or we reach the top (an empty branch).

t^ Start from the current branch of the Task tree. Search by climbing upward through the Task tree until either the target is found or we reach the top (an empty branch).

These strings can be combined to some extent. For example u^t^ means “climb up the User tree and then, if necessary, climb up the task tree”.

This ability to specify the search pattern for a file or variable is tremendously powerful. For example we might begin to include files with the title “hints.html” in the task tree. If each course, topic and problem node has a file called “hints.html” then we could specify it with simply @(t){hints.html}. But what if some nodes do not have a hints.html file? We could instead specify @(t^){hints.html}. This would insert the first file called “hints.html” found by searching upward from the current Branch. This means that if there is no hint file associated with this problem, we return the one associated with this problem set. Or if there is none for the problem set, we return the one for the whole course.

To take the example further, suppose that we determine that a particular student has a weakness in a particular subject. We could write a file called “hints.html” and put it in the user’s tree at an appropriate place. This particular user would then see the custom file when working at or below that level.

It will be seen from this example that it would be nice to be able to specify the hint file using @((ut)^){hints.html}, meaning “search in the User node, then the Task node, then climb up a level...”. Unfortunately this syntax is not currently supported.

There is a danger in using the “tree climbing” notation indiscriminately. If you are not careful you may end up giving a student access to a file you did not expect to. For example, if you inadvertently insert a file called @(t^)(solution.html) then this could be the solution to the current problem, or of any problem above this level in the current Branch!

6.3 The header and footer

If you have the served HTML version of the body.html document open in Netscape you’ll notice that it does not start with the words “Welcome Administrator”. It actually starts with a row of anchors and a horizontal rule. You’ll also have noticed that a similar header appears at the top of every page served by FlyingFish.

When FlyingFish processes a body.html document it actually performs three operations:

1 Process the file /header.html

2 Process and append the file body.html

3 Process and append the file /footer.html.

Note that /header.html and /footer.html are given as absolute paths (with the leading “/”, remember?). This means they are found at the root level of the task tree = (FlyingFish root)/task/

Open the file /header.html with a simple text editor. You will see something like this:

```
[@(t^){myHeader.html}]
```

The master /header.html file is nothing more than a reference to another file. The tag @(t^){myHeader.html} searches upward from the current branch until a file with the name myHeader.html is found. This means that different parts of the task tree (for example, different courses) can have different headers.

Here's an example of a typical myHeader.html

```
<HTML>
<HEAD>
<TITLE>@(u){/user/Name/Preferred}:  @(t){/branch}</TITLE>
</HEAD>
<BODY BGCOLOR="FFFFFF">
<TABLE><TR>
["@(t){/branch}"!="/"?<TD><P><a  href="../">Up</a></TD>]
<TD><A HREF="?Cmd=LOGO">Logout</A></TD>
</TR></TABLE>
<HR>
```

Note that the file begins with the <HTML> and <HEAD> tags. It also makes use of the variable insertion mechanism to set the title of the HTML document. @(t){/branch} is a special variable which gives the current Branch the user is looking at (e.g. /MS451/Kinetics/PQ_3/). It can be used on any FlyingFish page to give the user information about position in the problem set.

You should feel free to modify the /header.html or /footer.html documents. It is the best way to learn! If you make a mess of it and the files no longer work properly, you can always get a fresh copy from the CD. It is more usual, however, to edit the files myHeader.html and myFooter.html.

6.4 Conditional statements

Consider the following fragment from the file task/body.html:

```
[@{user/Login}<2  ?
This is the first time you have logged in. <br>
<i>Hajimemashite!</i> (Which is a Japanese greeting to someone you have never met
before.)<br>
:
You have logged in @{user/Login} times.<br>
]
```

This is a conditional statement which has been broken across several lines to make it easier to read. The structure is

[Condition ? Insert if condition is TRUE : Insert if condition is FALSE]

It is also acceptable to have only

[Condition ? Insert if condition is TRUE]

The condition is evaluated as a boolean expression and you can use the usual range of operators and delimiters. Note that the keywords "true" and "false" are NOT recognised; use integers 1 and 0 instead.

Comparison operators

Numerical relations:

equality a = b; a == b; a != b

inequalities a < b; a <= b; a > b; a >= b

String relations

equality “string1” == “string2”

inequality “string1” != “string2”

Boolean operators

or *expression1* || *expression2*

and *expression1* && *expression2*

not !*expression*

brackets ( )

In some future version of FlyingFish it will be possible to do arithmetic operations as well as comparisons. However these are not supported at the moment.

Examples of acceptable condition expressions:

```
(“string1” == “string2”) || 1 != 0
“@(u){user/Auth)” == “Developer”
2 3
```

Note the use of quotation marks around a variable identifier. The variable is inserted BEFORE the expression is evaluated so the variable value will also appear in quotes and can be compared as a *string*. If the quotes were left off then it would be a syntax error because we would be asking for an illegal comparison like this:

```
Student == “Developer”
```

or even

```
Developer == “Developer”
```

The order of precedence of the various operators is based on that in C, C++ and Java. Brackets are evaluated first, followed by negations, equalities and inequalities, followed by AND and OR. For absolute clarity, make good use of brackets.

A test document containing a range of acceptable and unacceptable conditional statements can be found in the standard installation. Follow the links to

Authoring / test_area / expressions /

A printed form of this test file can also be seen in Appendix 4.

6.5 Inserting an entire document

It is very useful to be able to insert an entire document into the text of another. The syntax works like this:

If you just want the text of the document in RAW FORM i.e. not processed:

@(modifiers){path name}

If you want the inserted file to be processed before insertion:

[@(modifiers){path name}]

The curly braces { } are not always necessary but it does not hurt to get into the habit of using them. They make it very clear where the path name starts and ends.

As an example of the usefulness of inserting a file, consider the following body.html document:

```
<H2>@(t){/branch}</H2>
<p><strong>Note that the radius of the path was not given on the printed sheet. Many apologies for this omission.</strong></p>
<CENTER>

<applet @(t){/codebase.html} code="Kinetics7100.class" width=500 height=180>
[@(t)/appletvital.html]
</applet>
</CENTER>
<P><font color=red> This is an assessed question worth a total of @(t){gene/marks} marks.</font></P>
@(t){/forumAnchor.html}
<BR>
```

This document is for MS451/Kinetics/AQ_2/. It is an assessed problem of the Java-applet type. The details of this type are given in a later section. For now, consider the line

<applet @(t){/codebase.html} code="Kinetics7100.class" width=500 height=180>

The tag @(t){/codebase.html} inserts a file from the task root called “codebase.html” which contains the full path name to the Java code base on the server computer. If for some reason this path needs to be changed we can change it ONCE in the master codebase.html file rather than 200 times in each of the HTML files for the individual java applets.

The tag [@(t)/appletvital.html] inserts a processed file from the task root called “appletvital.html”. Note that the processed file is inserted in the applet parameter list. It gives us a way of passing some parameters to each applet that is instantiated, in a flexible, extendable way. If each applet needs some new piece of information about the user or the environment, we can simply modify the /appletvital.html file:

```
<!-- this HTML fragment is parsed and passed on to each JellyfishClient applet
DON'T EDIT THIS FILE unless you really know what you're doing! -->
["@(u){/status/User/Auth}"="Developer"?<param name=diagnostic value=TRUE>]
<param name=User value=@(u){/status/User}>
```

```
<!-- it is polite to always have a message that is shown if the user does not have Java
- - >
You should be seeing a Java(TM) applet here. Either your Web Browser does not
support Java or it is turned off at the moment.
```

Note that the process of inserting a file can be recursive in the sense that a file can insert itself, or there can be a series of insertions that form a closed loop. This is one of very few ways in which you as a content author can cause the FlyingFish server to behave badly. The server does not currently test for circular insertions and it would cheerfully continue inserting files into themselves until all available resources were exhausted. Then it would crash.

6.6 Testing for the existance of a file or UDB value

Sometimes it is helpful to be able to use a conditional statement to determine if a file or UDB value exists. The syntax is very similar to the syntax for inserting the file or variable:

- To insert a variable or file we write @(modifiers){path}
- To test whether the variable or file *exists* we write @?(modifiers){path}

You can think of the second version as returning either 0 or 1, depending on whether the variable or file exists. So we can construct conditional statements of this form:

```
[!@?(u){/user/Name/Preferred} ? You have not entered a Preferred name]
```

The words “You have not entered a Preferred name” will only appear in the final HTML file if the condition evaluates to TRUE i.e. if there is NOT a UDB value (/user/Name/Preferred).

6.7 Escape characters

You now know that, in the FlyingFish Environment, the @ symbol is “special” since it always introduces a variable or file insertion tag. But the @ symbol is useful for other purposes as well, for example when giving an email address. If a file contained the following email address:

```
nscott@mech.uwa.edu.au
```

FlyingFish will treat it like this:

```
nscott[try to insert variable mech.uwa.edu.au by relative reference]
```

Naturally this variable will not be found so nothing will appear after “nscott”.

To get around this problem, simply “escape” the @ symbol by putting a back-slash character (\) in front of it, like this:

```
nscott\@mech.uwa.edu.au.
```

The @ symbol is not the only character to watch out for. If your HTML is not looking right it could well be because you have forgotten to escape one of the following:

TABLE 2 Special characters[10]

Character	*Meaning*	*Example*
@	Substitute a variable	@(u){/user/Name/Family}
:	ELSE in conditional statement	["martini"=="dry"?TRUE:FALSE]
\	Escape character	To show an @ symbol use \@

Any of these special characters can be shown on the HTML page – all you have to do is to put the escape character immediately before it (e.g. \:). If you want to show the escape character itself, use two of them in a row, like this: \\. This approach to special characters is analogous to the approach taken in the computer languages C, C++ and Java.

When I say that a character is "shown" I mean that the Web Browser will receive the character, i.e. it will appear in the file you see if you select "view source" from your Browser menu. In a typical HTML file there are characters that are not intended to be seen by the user - anything in a < > delimited tag, for example.

Note that there are cases where special characters, including the escape character, will NOT be respected and will thus appear in the final HTML that the user sees. This can be a useful behaviour if you need to substitute some JavaScript into a page. See the section on the mapping file, below.

It is a good practice to always escape the @ symbol because it is always interpereted as a variable or file reference. It is not so clear whether the : symbol should always be escaped. The preprocessor actually ignores the ? and : symbols if they appear in the Jellyfish source HTML outside a preprocessor bracket pair []. The problem is that you might write some HTML that contains the : character and then later on add some conditional statements that effectively place the HTML into square brackets. Any un-escaped : character can suddenly take on unwanted significance and will tend to make your IF-THEN-ELSE statement behave unexpectedly. So it is best to escape them all and avoid trouble.

6.8 Date strings

Date information can be stored in the form of a string, e.g.:

---

10 Earlier versions of the server defined some additional special characters:

Character	*Meaning*	*Example*
#	Shorthand notation for input UDB in user tree at this level	#answer is shorthand notation for @(u){input/answer}
$	Shorthand notation for state UDB in user tree at this level	$done is shorthand notation for @(u){state/done}

These definitions were made for a specific course and proved unworkable in some other courses, so they are no longer supported. HTML containing escapes for these e.g. \# will continue to work.

```
Thu 31 December 1999 6:00PM
```

The order of the elements in a date string is not important, and upper or lower case are acceptable.

Smart date correction

If the day of the week is given in a date string, it has a higher significance than the day of the month. In cases where the date is internally inconsistent (as in the above example – the 31 Dec 1999 was actually a Friday), the date is adjusted to make the day of the week correct. In other words, the above example will be corrected internally to

```
Thu 30 December 1999 6:00PM
```

Missing date elements

If any of the elements of a date string are missing, the *current server value* is substituted instead. This substitution behaviour can This is actually a very useful behaviour. Consider the date string

```
Fri 20 May 6:00PM
```

The year is not specified. So the server will insert the *current* year value e.g. 2000 or 2001. Furthermore, since the string *does* have the day of the week, the server will choose the Friday in the current year *closest to the 20th of May*. If you wish you can exploit this behaviour while setting your deadlines. Set them like the above example and they will be correct – or very nearly correct – every year without adjustment.

6.9 Functions

There are cases where insertion of a file or variable is not enough; sometimes we would like to insert something which is *calculated* from other known values. Examples include

- The deadline for a particular user, when a random “staggering” of dates is desired;
- The current date and time at the server;
- A very elaborate HTML tag which would be awkward to program using insertions and conditional statements.

In this system release, functions are hard-coded in the server binaries. This means that you can’t add new functions and you can’t change the ones listed below. In some future release it may be possible to “plug in” additional function modules.

The syntax for invoking functions is somewhat idiosyncratic. Some functions look like UDB variables e.g.

```
@(t){/branch}
```

is a function that returns the current branch. All such “ghost variable” functions are “in the task tree” and should be referred to by absolute reference

(hence the leading slash before /branch). These “ghost variables” are supported because they were defined in an early system release and are now widespread. In other words, although we would like to eliminate them, too many users would be upset!

The modern type of FlyingFish function uses the C, C++ and Java convention of round brackets. Functions can be “stacked” e.g.

```
@(t){value.func1().func2()}
```

This means that the value is modified by func1 and then the modified value is modified *further* by func2.

Some functions can be used in several ways, e.g.

```
@(t){ticks()}
```

is the current ticks() value at the server;

```
@(t){deadline().ticks()}
```

is the value of the date string deadline() converted to a ticks value; and

```
@(t){ticks(9:49AM Wed 01 Dec 1999)}
```

converts the date string to a ticks value.

Some functions are context-sensitive. For example the date() function, applied to a date string, simply cleans it up according to the section above on date strings. However you can also apply the date() function to a ticks value and the result will be what you expect.

Expression	Notes	Result
@(t){/branch}	/Authoring/test_area/functions/	/Authoring/test_area/functions/
@(t){/trim_branch}[11]	/Authoring/test_area/functions/	/Authoring/test_area/functions/
@(t){/version}	The version number of the server	v1.39
@(t){deadline()}	The deadline for this task in a standard human-readable form	6:00PM Mon 30 Dec 2002
@(t){deadline().ticks()}	The deadline for this task as a number of seconds since 1 Jan 1970	1041271200
@(t){ticks()}	The time (according to the server) as a number of seconds since 1 Jan 1970	947591551

11 This will only be different from /branch if the current file is NOT body.html

@(t){ticks().date()}	The time (according to the server) in human-readable form	11:52AM Tue 11 Jan 2000
@(t){ticks(9:49AM Wed 01 Dec 1999)}	The ticks() function can also convert a human-readable date into a number of seconds, which is useful when you want to compare two dates.	944041740
[@ticks() < @ticks(9:49AM Wed 01 Dec 1999)?Before:After]	An example of a date comparison	After
@(t)date(944041785)	The date function can also process a "ticks" value passed as an argument	9:49AM Wed 01 Dec 1999
@(t)date(9:49AM 1 Dec 1999)	The date function can also process a date value passed as an argument. The date is cleaned up, completed if necessary, and presented in the standard format.	9:49AM Wed 01 Dec 1999
@(t){option_tag(test/tag_value, Fish, Bird, Elephant, Goat)}	A popup menu allowing a selection, stored in the task tree	
@(u)option_tag(test/tag_value, Camembert, Swiss, Gouda, Parmesan, Limberger)	A popup menu allowing a selection, stored in the user tree	
@(u){/user/Authority.AuthInt()}	The authority level of the *being* user as an integer between 0 and 50	50
@(a){/user/Authority.AuthInt()}	The authority level of the *actual* user as an integer between 0 and 50	50

7. Special files

In an ordinary web site, all files are available for viewing (provided a user knows what location to type into the browser). In the FlyingFish system there are some additional files in the task tree which are not meant for direct viewing, but which govern a range of behaviours including

- Setting deadlines and available marks for tasks
- Setting access restrictions
- Setting some system preferences
- Setting MIME type mappings

And so on. This section explains some of the more important special files used by the FlyingFish.

Note that it is OK to define your own additional files, and it is also OK to add additional keys to these "special" files. The UDB file format is deliberately extensible and open, and your additional keys will not affect the existing ones. There is a small chance that conventions you define for your own purposes could conflict with future system extensions – but we will deal with this when it happens.

7.1 Course requirements and limited access

In a sense the FlyingFish Tutorial Environment serves just three purposes:

1 To keep records of user activity;

2 To pose a series of problems for the user to solve; and

3 To prevent the user from gaining unauthorised access to some resources.

Ironically, good teaching seems to involve making some information patently obvious while at the same time *hiding* some information until an appropriate moment. For example, we want the equation sheet for a course to be available at all times to the students, but we want to hide the solution to an assessed problem until after a certain date.

In this section we will explore points (2) and (3). How do we specify which problems the user MUST complete? How do we limit access to some resources and then grant access at an approriate moment?

Each node in the Task tree can have a UDB file called simply "info". This file can contain certain key=value pairs that tell FlyingFish how to view the node. Remember that a node is either a single problem or a group of problems such as an entire course. Here is the info file from /MS451/Kinetics/:

```
Access/Condition="@(u){/MS451/Kinematics2/Done/DONE}" == "1" ||
"@(u){/status/User/Auth}"="Developer"
Access/Denial=You have not yet finished all the problems in the previous set.
Complete=PQ*.*, AQ*.*
Deadline=Wed 19 Aug 1998 5:00PM
```

The meaning and effect of the Access, Complete and Deadline entries is explained below.

7.1.1 INFO/ACCESS

The info/Access key can have two sub-keys:

info/Access/Condition is a boolean expression which is evaluated to determine if the user is allowed to see the body.html document at the same level as this info file.

info/Access/Denial is a message the user will see if access is denied, usually a brief explanation of the problem.

In the UDB file given above, Access is granted if either "@(u){/MS451/Kinematics2/Done/DONE}" == "1" (meaning that the user has completed the previous problem set), or if "@(u){/status/User/Auth}" =="Developer", meaning that the user has a certain authority level.

You may think it odd that we test the completion of the previous set using string comparisons. The problem is that the Done/DONE key only appears when a node has been completed, and then it always has the value "1". If the key does not exist at all then it has no value, neither 0 nor 1, so an integer-based comparison will sometimes produce a syntax error and fail.

The info/Access/Condition values can be anything you like. In this case they are being used to force the student to complete some problem sets in a certain order. Each problem set except the first has a Condition which names the previous set.

There are cases where we wish to grant the student access to a node without necessarily satisfying the info/Access condition. This mechanism is used to control access to numerous resources associated with Java applet questions. FlyingFish will unconditionally grant access to a node if the user has a UDB file called "Access" in their user tree at the same level. These files are created by FlyingFish in response to certain signals from a Java applet but not under other circumstances. So access to a given node is granted if:

1 The user has an Access file for the node; or

2 There is no info file for the node; or

3 The info/Access/Condition expression evaluates to TRUE

7.1.2 INFO /COMPLETE

If a given node contains twenty sub-nodes, how does FlyingFish know which (if any) of these are sub-tasks that must be completed?

The info/Complete value of a node specifies a list of sub-tasks (i.e. sub-directories or sub-nodes) which must be deemed complete before this node is deemed complete. The list can be explicit, for example

```
Complete=PQ_1, PQ_2, PQ_3, PQ_4, AQ_1, AQ_2
```

But this could become very tiresome if there are more than a few sub-nodes. So a "wild-card" notation is allowed. The list

```
Complete=PQ*.*,  AQ*.*
```

Means "all nodes with names matching 'PQ*.*' AND all nodes with names matching 'AQ*.*' must be completed". The order of the elements in the list has only a very minor effect on the experience of the student. All matching nodes

must be completed, regardless of the order. The order determines only the "suggested branch", which is the branch we present to the student as something to go on with. If the last node matching "PQ*.*" has been completed, FlyingFish will then suggest that the student should next attempt the first node matching "AQ*.*.

7.1.3 SOLUTION NODES

A common problem is limiting, and then granting, access to the solution to a given question. It is a problem because students are not ideal, mature adults, but suffer from social-learning difficulties that make them do strange things. For example, they have been known to use the solution to a problem to "get through", without necessarily understanding how or why the solution works. In setting assessed work we are continually confronted with students who "pass around" the solution to the work and gain credit unfairly.

Over the years we have developed some strategies to discourage these inappropriate working habits and encourage productive ones. Some strategies that have been found useful are:

- To prevent a student from seeing the solution to a problem until he or she has had a reasonable attempt at it (and still needs help).
- To hide the solution to all assessed problems until the deadline has passed, whether the student completes the problem or not. This is to prevent "traffic" in printed solutions.
- To have "staggered" deadlines so that students can get easy access to computing resources, and thus are not forced into opportunistic work modes.
- To show all solutions to all problems – whether the problems are completed or not – once the deadline has passed. This is to help students who have fallen behind. Usually there is no credit available anyway.

Solution nodes are protected by the same info/Access/Condition mechanism as all other nodes and there is nothing special about them. However it is often the case that we want a great many problems to follow the same rules for hiding or showing the solution. One way to do this is to have a single UDB file somewhere high in the task tree, called (e.g.) "solution_access":

```
p/Condition=[(@(t){deadline()/time()} < @(t){time()}) ||
    @(u){/user/Authority.AuthInt()} > 10
p/Denial=[The solution to this practice question will become available
    @(t){deadline()}]
a/Condition=[(@(y^){info/deadline/time()} < @(t){time()}) ||
    @(u){/user/Authority.AuthInt()} > 10
a/Denial=[The solution to this assessed question will become available
    (@(y^){info/deadline/date()}]
error/Condition=@(u){/user/Authority.AuthInt()} > 10
error/Denial=This error message is only made available when you make a specific
    error
```

Then, to protect the solution to a practice question, the info file looks like this:

```
Access/Condition=[@(t^){solution_access/p/Condition}]
```

```
Access/Denial=[@(t^){solution_access/p/Denial}]
```

And to protect an assessed question, it is

```
Access/Condition=[@(t^){solution_access/a/Condition}]
Access/Denial=[@(t^){solution_access/a/Denial}]
```

Note that the tree-climbing syntax is used in the info files – this means that you can put a different solution_access UDB file into a problem set and it will override the higher-level one. You have control over several hundred solution info files and you *also* have the ability to make some problems or problem sets obey a different rule.

7.2 Deadlines: info/Deadline and info/Deadline/stagger

The deadline for a Node is implicitly the deadline for completing all the sub-nodes. The significance of the deadline depends on other qualities of the sub-nodes, for example whether any marks are attached to them. Currently FlyingFish will only award marks for assessed problems before the deadline. In future versions there will be additional info node tags to modify this default behaviour.

Here is an example of a Deadline key in an info file:

```
Deadline=Fri 20 May 6:00PM
```

You can create an info file with a Deadline key in any of the following places:

- In the user tree for a particular user - a private deadline for one student;
- In the year tree; or
- In the task tree.

The reason is that the standard search pattern for deadlines is like this:

```
@(u^y^t^){info/Deadline}
```

It is probably a good idea to choose either the year tree OR the task tree, and put all your deadlines there to avoid confusion.

Staggering and the deadline() function

Students tend to leave work until the last minute. This means that in a large class, your computer lab(s) can be overcrowded on the days that work is due. We have found a simple technological solution to this common problem: to stagger the deadlines by a few days. Staggering means that some students have a deadline which is earlier than others. The amount of stagger is random so that students are not always first or last.

To set a stagger value, you have to put an info/Deadline/stagger key in either the user, year or task tree. This key can be at any level: high up as a default value, or deep in some course material. The stagger key is found using the search pattern (u^y^t^), just like the info/Deadline key.

Consider a course called Dynamics which has a problem set called Rectilinear. We might put files in these locations:

Tree	File	File contents	Explanation
y	/Dynamics/info	Deadline/stagger=3	See below *.
t	/Dynamics/Rectilinear/info	Deadline=Fri 24 Sep 6:00PM	Sets the deadline for the Rectilinear problem set.

* We want student deadlines to be randomly "staggered" by 3 days so that the computer room is not too crowded. Set this to 0 or delete it to give all students the same deadline. A different stagger value can also appear lower in the tree if you wish – it will “override” the higher value because of the “climbing” searches (u^y^t^).

Note that the stagger value can be read as “the number of consecutive working days” where deadlines will occur. If

Example: showing a deadline to the student

The obvious way to show a deadline value on a page is to insert it as a variable in the usual way:

```
@(u^y^t^){info/Deadline}
```

However this is not ideal because we would see the literal value of the info/Deadline key - which might be incomplete or even inconsistent. It is better to use the deadline() function

```
@(t){deadline()}
```

Note that this is not a UDB value - it is a function call. The FlyingFish creates a deadline string which is corrected for the student's current time zone and may also be staggered (if you have set a stagger value).

Sometimes it is helpful to show students deadlines for other parts of the tree. It is easy to do this – just use a path before the deadline() function, like this:

```
@(t){../deadline()}
```

or

```
@(t){Curvilinear_XY/deadline()}
```

Example: testing whether a date has passed

It is useful to customise HTML pages based on the time they are seen. For example, you might want to hide certain information (such as the results of a test) until a certain date has passed. It is easy to do this using a combination of the conditional statment and function notation. The trick is to convert the "human readable" date strings into integers before doing the comparison:

```
@(t){deadline()}
```

returns a string like "6:00PM Fri 24 Sep 1999"

```
@(t){deadline().ticks()}
```

returns a string like "93320733", which is the number of seconds since 1 Jan 1970.

```
@(t){ticks()}
```

by itself returns the CURRENT ticks, i.e. the ticks at the time the HTML document was generated.

So you can test whether the deadline has passed using a conditional statement of this kind:

```
[@(t){ticks()} < @(t){deadline().ticks()}?Before deadline:After deadline]
```

7.2 User filter: info/Student

A typical FlyingFish installation serves many courses from a single FlyingFish instance. As a result there are users who belong to unrelated class groups in the system. Yet when we want to see the progress of the class we only wish to see the progress of students who are assigned to that class. FlyingFish provides a very simple, but effective, solution to this problem.

At any level of the task tree there can be a UDB key

```
info/Student=[Conditional statement]
```

The conditional statement should be a fragment of HTML that, when preprocessed in the context of a given user, will evaluate to either 1 or 0. For example:

```
info/Student=@?(u){/user/Course/600101}
```

Each user object has a chance to evaluate the conditional statement, and if it turns into something other than 0, the user is “in the list” for the course. In this case we are testing the value of a key in the main user file (@(u){/user...}) but you could test other keys or even use boolean operators like && and ||.

It makes sense to have an info/Student filter in each of your courses, and to have a logical system for the keys that you test. I have found it helpful to have the convention of only testing subkeys of user/Course, and the subkeys are the official course codes used by my University. It is then a good idea to provide staff with an easy way to turn the student’s “flags” on an off. I do this through the admin page, which has:

```
<INPUT NAME="/user/course/600101" TYPE="hidden" VALUE="null">
<INPUT TYPE="checkbox" [@?(u){/user/course/600101}?CHECKED] Value=1
NAME="(u)/user/course/600101"> Engineering 101
```

When processed and displayed by the browser, this turns into a checkbox which will “switch” the value of the user variable /user/course/600101 on or off.

Usually I surround statements of this kind with a test on the authority level of the *actual* user, so that students do not see the checkbox[12]:

```
<TABLE>
[@(t){/authority}<20?
<!-- students see this version -->
<TR><TD>User ID\:</TD><TD>@(u){/user/UserID}</TD></TR>
<TR><TD>Surname\:</TD><TD>@(u){/user/Name/Family}</TD></TR>
<TR><TD>First name\:</TD><TD>@(u){/user/Name/Given}</TD></TR>
:
<!-- staff version -->
<TR><TD>User ID\:</TD><TD><INPUT TYPE="text" NAME="/user/UserID"
VALUE="@(u){/user/UserID}" SIZE="20"></TD></TR>
<TR><TD>Surname\:</TD><TD><INPUT TYPE="text" NAME="/user/Name/Family"
VALUE="@(u){/user/Name/Family}" SIZE="20"></TD></TR>
<TR><TD>Given name\:</TD><TD><INPUT TYPE="text" NAME="/user/Name/Given"
VALUE="@(u){/user/Name/Given}" SIZE="20"></TD></TR>

<TR><TD>Courses\:</TD>
<TD><INPUT NAME="/user/course/600101" TYPE="hidden" VALUE="null">
<INPUT TYPE="checkbox" [@?(u){/user/course/600101}?CHECKED] Value=1
NAME="(u)/user/course/600101"> Engineering 101</TD></TR>
]
</TABLE>
```

7.3 Mapping files: setting MIME behaviour

When you look at a web page you are actually seeing something generated by a web browser based on one or more files. The source files typically include:

- An HTML page which provides text and can also include embedded pictures, sounds, movies and so on.
- An additional file for each picture, sound, or movie.

It is worth remembering that, because of this, each picture you include on a web page produces a separate “hit” on your web server.

In its original form the Web was text-only, but over the years a great range of additional file types have become accepted and widely used. The most common are pictures (GIF and JPEG format mainly) but there are now hundreds of supported file types. To help reduce the chance of confusion, with data of some kind being mistaken for data of another kind, there is a system of “file type naming” called MIME[13]. MIME is actually a huge set of standards and is very flexible, but it can also be used as a way of specifiying the “content type” of a block of data. Well-behaved web servers are supposed to send a MIME type

---

12 Note that, even if this control were unintentionally shown to students, this would NOT mean that students could change the setting. By default students cannot change any variable values, unless there are special *permission* UDB keys in the task tree.

13 See Borenstein, N and Freed, N, Request for Comments 1341, “MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies”, June 1992

along with every block of served data, so that your browser will know exactly what it is receiving. So the question is, how does the FlyingFish know what MIME type to return with every file?

To make this discussion more concrete, consider the following URL:

```
http://dynamics.mech.uwa.edu.au/jellyfish/baking_230/myPicture.GIF
```

This is a request for a GIF file. The FlyingFish server at the given address will return the contents of the file called myPicture.GIF which is (hopefully) in the directory [task]/baking_230/. But it must also send a MIME type along with the picture file data. What MIME type will it send?

The answer is that the FlyingFish makes use of UDB files with the name "mapping", which can live anywhere in the task tree. Here is an example of a mapping file, with large chunks deleted (...) to save space:

```
.arj=application/x-arj
.au=audio/basic
.avi=video/msvideo
.bin=application/octet-stream
.bmp=image/x-windows-bmp
...
.htm=text/html
.html=text/html
.html/header_footer=1
.html/process=1
...
.js=application/x-javascript
.js/no_escape=1
...
.txt=text/plain
.txt/header_footer=1
...
default_file=body.html
```

Most of the lines in the file are of the form

```
.xyz=MIME type
```

where ".xyz" is the file suffix of a file type on a Windows™ computer. For example, if the FlyingFish is asked to return a file called "thing.txt", it will also return the MIME type found using the search

```
@(t^){mapping/.txt}
```

i.e. (using the mapping file above) MIME type "text/plain".

The search (t^) makes it possible to "override" a high-level mapping file by putting another one at a lower part of the task tree i.e. in a subdirectory. Note that you don't have to copy all the keys to the overriding mapping file – only the ones you are changing.

mapping files can also contain some additional keys. All of the possible keys are in the example mapping file above:

key	effect
.xyz/header_footer=1	Files of type .xyz are returned* with the @(t){/header.html} and @(t){/footer.html} attached.
.xyz/process=1	Files of type .xyz are processed* as though they were Jellyfish-enhanced HTML. This means that substitutions are performed and the escape character is meaningful.
.xyz/no_escape=1	The escape character will be ignored in files of this type, and will appear in the final HTML output. This is essential to the proper functioning of JavaScript fragments, because the escape character is used as in Java itself, and must not be interpereted or removed.
default_file=index.html	Each directory in your task tree can define its own default-file name.

* Top-level file only – see below.

Note that sub-keys of a file type, such as .xyz/header_footer, are NOT found using the obvious search pattern

```
@(t^){mapping/.xyz/header_footer}
```

instead they are found by first finding

```
@(t^){mapping/.xyz}
```

then the subkey is expected to be immediately below the found UDB. This approach is necessary so that it is possible to turn “on” the sub-key options (such as header_footer), and also to turn them OFF at a lower level.

DEFAULT_FILE

The mapping file line “default_file=something.html” deserves more attention. Recall that in a previous chapter the idea of a *default file* was explored. Specifically, it was explained that a URL of this form

```
http://dynamics.mech.uwa.edu.au/jellyfish/baking_230/
```

is interpereted to mean “return the default file in the directory baking_230”. The default file name for FlyingFish is usually body.html. However the mapping file key default_file=[name] establishes a new default-file name convention which applies at the branch of the mapping file and in all subdirectories (unless they contain another mapping file which establishes its own convention). This was set up to assist content authors who already have large web sites that use a different default-file-name convention e.g. index.html. Such a site can be brought into the FlyingFish task tree without modification – all you have to do is put an appropriate mapping file in the top level directory.

MIME TYPE, PROCESSING OPTIONS AND EMBEDDING BY SUBSTITUTION

It will be seen that the mapping file allows precise control over MIME typing of data and also over some processing options, on a level-by-level or directory-by-directory basis. However there is one final subtlety that we must address. When a file is requested by a brower, there is no doubt about the MIME type of the returned file. If a text file is requested, the MIME type of the returned data is set to "text/plain", or whatever the mapping file dictates. But what if an HTML file is requested, and the requested file *subtitutes* a text file within it? In other words, what happens if the following body.html file is requested?

```
<P>This is HTML</P>
@(t){my_text_file.txt}
<P>More HTML</P>
```

We must make a distinction between the body.html file (the originally requested file) and the file which is "slurped in" during the substitution process. We will call the originally requested file the "top level file", and the substituted one(s) "subordinate". The rules are as follows:

- The default behaviour is that the subordinate file is inserted into the top level file and is subject to the same processing and escaping as the top level file. HOWEVER
- If the subordinate file type has the no_escape qualifier, this is honoured even during substitution. This means that (using the mapping file example above), .js fragments can be inserted into HTML without corrupting the escape character, even though the HTML top level file *will* be escaped as usual.

If this discussion has made your temples throb, don't worry. The standard FlyingFish installation comes with a good mapping file at the top level. This top mapping file contains MIME types for a huge range of file types, and sensible default settings for substitution behaviour. You only need to mess with the mapping file, or add extra mapping files, if you need some special behaviour or you have some bizarre new file type.

7.4 Permission keys

It has been wisely said that "obscurity is not security" with respect to computer software. In other words, we cannot rely on the user's ignorance to protect private data or system function. You must assume that your students have access to this document and have studied it as thoroughly as you have, and that they know HTML and CGI conventions completely. Such a user could edit the Location (URL) field of the browser to mimic any HTML form. For example, suppose that at the branch /Baking_230/ there is an HTML form:

```
<FORM METHOD=GET>
<INPUT TYPE=HIDDEN NAME=Cmd VALUE=STOR>
<P>Your mark for this course:
<INPUT TYPE=TEXT SIZE=5 NAME="(u)Done/AVTM" VALUE="@(u){Done/AVTM}">
<INPUT TYPE="submit" NAME="ignore" VALUE="Submit"></P>
</FORM>
```

If the user enters “10” and presses the Submit button, the browser will request the page with the following URL:

```
http://130.95.52.28:8080/jellyfish/~soybomb/Baking_230/?Cmd=STOR&Done
%sFAVTM=10&ignore=Submit
```

The %2F combination is the encoded ‘/’ character, which cannot appear in a query string. The purpose of the STOR command is to store data from an HTML form, so the above form – or emulated form – will in theory edit the value of the student’s mark for the Baking_230 course[14].

The point here is that a knowledgeable user can emulate any HTML form at all, by typing the right string into the Location field of the browser, and can attempt to set user-tree or even task-tree variables. We cannot rely on students not knowing about this, particularly if you use METHOD=GET in your forms, since the GET method reveals the query string in the Location field of the browser.

In order to provide real security, we have to have a system that will do the right thing even if improper requests are made e.g. a student changing his or her own mark for an assessed question. FlyingFish uses the authority level of the actual user (the one who gave a password) to determine whether information is to be believed when it comes in:

- Users at Helper level and above can modify any user or task UDB without question. A Helper-level user submitting the form above really could set the student’s mark for the Baking_230 task, although the STOR command is not the correct way to do it[15].
- Users at Guest and Student level *cannot* modify any user or task UDBs unless specific permission is granted.

There are times when we *do* want Student-level users to have control over a user variable. For example, consider the admin page where users can set their preferred name:

```
<FORM METHOD=POST>
<INPUT TYPE=HIDDEN NAME=Cmd VALUE=STOR>

    [... lots of other input fields ...]

<P>Preferred name:
<INPUT TYPE="text" NAME="/user/Name/Preferred"
VALUE="@(u){/user/Name/Preferred}" SIZE="20"></P>

    [.. more input fields ...]

<INPUT TYPE="submit" NAME="ignore" VALUE="Submit">
</FORM>
```

14 Don’t panic – students can’t really change their own marks. Read on.

15 The STOR command just stores things; it is not smart enough to recalculate higher-level marks. Mark changes are accomplished with the MARK command, which is effectively a STOR that also invokes the standard Done mechanism at the current level.

Using the rules given above, students might well enter a preferred name e,g, “tamlin” and press the Submit button, but the form would come back unchanged. FlyingFish would reject the proposed variable change (u)/user/Name/Preferred=tamlin. To allow Student-level users to have control over a user variable, we have to specifically grant permission to change it by creating a special permission UDB in the task tree. For this example the permission key would be in a file

```
[task  root]/user
```

and the file would contain a key

```
Name/Preferred/permission=u
```

The standard release version of FlyingFish has a [task root]/user file containing

```
Name/Family/permission=u
Name/Given/permission=u
Name/Preferred/permission=u
Password/permission=u
Sex/permission=u
Title/permission=u
email/permission=u
PrivateWeb/permission=u
```

Because we grant the student the ability to change all of these fields.

To grant a user permission to alter data in the task tree, the permission key is

```
permission=t
```

It is also possible to stack up the permission flags e.g.

```
permission=uyt
```

would allow Students to make changes to some UDB value in the user, year or task trees.

It would be unusual to allow student users to change variables in the task tree, mainly because the data could then be subject to rapid, repeated changes. If several hundred users are all getting and setting the value of some variable, then its value at any instant might not be very meaningful. However, the facility is there if you can think of a use for it.

8. Detailed FlyingFish configuration

The FlyingFish package is designed for use by ordinary university teachers, and the release version contains configuration files to allow almost anyone to quickly have something running. In Chapter 2 (Installation) the sequence of steps for installation is almost nothing more than “decompress, run”. In fact, FlyingFish will even run off a CD-ROM, where no configuration is possible at all (although no user data can be saved either). This section is only for people who want to set up something unusual and therefore must change the jellyfish.ini file.

8.1 The jellyfish.ini file

Here is an example of a jellyfish.ini file:

```
tree/u=../user
tree/t=../task
tree/y=../2000
farm/address=130.95.16.27
farm/port=5555
http/port=8070
admin/email=nscott@mech.uwa.edu.au
admin/name=Dr Nathan Scott
timeout/idle=1800
GMT Offset=-28800
persistent/mozilla=1
persistent/msie=1
license=1anz-a8rq-9y4b
license/expires=31 Dec 2001 11:59PM
license/hostIP=130.95.52.241
license/software=FlyingFish
```

It is a UDB file and is unusual only because, unlike other UDB files, it has the file type “.ini”, instead of the usual empty file type. FlyingFish looks for the jellyfish.ini file in the same directory as the FFishXXX.exe executable. If it is missing, FlyingFish will not start at all and there will not even be an error message.

Each part of the jellyfish.ini file will be explained in a separate section below.

8.2 tree

The tree subkeys define the mapping between the internal trees used by FlyingFish and the computer file system. The minimal set is

```
tree/u=[path]
tree/t=[path]
```

However it is common (but optional) to also define the year tree

```
tree/y=[path]
```

It is also possible to define any number of additional trees, e.g.

```
tree/x=[path]
```

And these trees can be used to store files or variables. To refer to a variable in the x tree from HTML you would use a substitution expression like this:

```
@(x){/path1/path2/filename/var1/var2}
```

FlyingFish makes no great distinction between the "usual" trees (u, t and y) and "custom" trees such as x. All are managed by the same internal mechanisms e.g. UDB object trees and file caches. However, it would be unusual to need a custom tree like x.

Each tree/* key has a value which defines where the root of the tree is. Two conventions are accepted:

- Absolute file system references, e.g. tree/t=d:\FlyingFish\task
- Relative references, e.g. tree/t=../task

The advantage of the relative form is that, provided you keep the same set of directories in your FlyingFish installation, the references will always be correct, even if you move the entire FlyingFish directory to another hard disk. Relative references make it possible to put FlyingFish on a CD-ROM – it will still work, right off the CD, regardless of the "drive letter" of the CD drive on the particular computer you are using. Of course, if the user directories are also on the CD-ROM, then user progress data cannot be saved – but this is OK if you are only trying to produce a demo. I burn a CD version of a site before giving a presentation about it: this guarantees that every time I start it up, it will behave exactly like the previous time.

It is possible to have several instances of FlyingFish on the same host computer (see http/port, below), and these instances can even share some files on the hard disk. For example, it is OK to have two copies of FlyingFish, each of which uses the same htdocs directory. It is also OK to share task directories and even user directories, although it would be unusual to want to do this. Note that, if files are shared by several instances of FlyingFish, there may be occasions where a file is updated by one instance but is not immediately "read in" by the other, as a result of file caching. If you have an application that seems to require simultaneous dual FlyingFishes, talk to Advance Education technical staff because there may be a simpler way to do the job.

8.2 farm

Trees? Farms? Fish? What next?

The farm is a suite of software developed by Dr Kevin Judd which allows FlyingFish to communicate with Mathematica™ kernels on another computer. At the time of writing it *must* be another computer because the Farm software is UNIX-based and has not been converted to run under Win32.

The keys identify the IP address and IP port number of the farm software. More detail about the use of the farm software can be found in the documentation for the Calmæth™ system.

8.3 http/port

This is the port number used by this instance of FlyingFish when it is running. You can use any port number you like in the range 0..65535 (with some conditions, see below).

Internet Protocol (IP) port numbers have been around for a few years now (since the early 1960s) and as a result there are some conventions which it is good to be aware of. The first is that, on a given computer, there can be only one *listening* port with a given number. If one program starts listening for connections on port 80, and then another program tries to start another listening port with the same number, the second program will get an internal error and will not be able to continue. This makes sense, because otherwise it would be like having two houses in one street with the same number: who gets the mail? Note that there can be any number of *connecting* ports (the more usual kind) with a given number. You have probably experienced this already because you have had many instances of Netscape Navigator™ running simultaneously, and each one usually connects on port 80.

Secondly, some port numbers are associated with certain web services:

Port	*Traditional web service*
0	Don't use this; I'm sure it is reserved for some obscure IP function. Besides, it's downright weird.
21	Telnet
80	HTTP (web)
8080	HTTP (web), if port 80 is in use. Also sometimes used by Proxy services
8888	Often used by Proxy services.

I am uncomfortable with the use of any low number as a port number, because I am paranoid about accidentally using "someone else's standard port". I make my port numbers either 80 or in the range 8000-8999. If you ever do accidentally use someone else's port number, it is not really all that serious as it can only affect services *hosted by your computer*. It might mean that some other server program on your computer (or the FlyingFish itself, if it happen to come alive second) will not be able to start properly. If you are in full control of the computer, you probably know what services are hosted by it and what port numbers the services want to use.

Sometimes you will want to use one of the "standard" port numbers. For example, if you are setting up an instance of FlyingFish to be the main web server on your computer, there is no harm in allowing it to use port 80. This will mean that the address of your FlyingFish will be simplified – web browsers assume that if the port qualifier (e.g. :8060) is missing, that port 80 should be used. In other words, if you run your FlyingFish on port 80, the user needs only to type

```
http://130.95.52.28/
```

to make contact with it (where the IP address is of course the address of your computer). This is slightly simpler than, e.g.

```
http://130.95.52.28:8080/
```

You can run any number of FlyingFish instances on one computer provided that each one has a distinct http/port number.

8.3 admin

The admin/email and admin/name keys should be the email address and name of someone who can fix server problems, or at least someone who regularly reads email. The FlyingFish knows this address internally as the "Server Admin" address, and it can be displayed using the special CGI variable reference

```
<A HREF="mailto:@(c){/CGI/Server Admin}">Send email to the WebMaster</A>
```

This address is also used in the (unlikely) event that an error has occurred internally but no error.html file can be found. In this case a hard-wired error message is shown, and it includes the admin/email address.

8.4 timeout/idle

If a user stops working without logging out, it is a good practice to "time out" the user after a reasonable interval. The timeout/idle value is the number of seconds FlyingFish will wait before logging a user out. We have found that 20 to 40 minutes is a good value for first-year students.

8.5 GMT Offset

Increasingly we have students who are distributed across several time zones. If such students all have a deadline at 6pm, server time, it would be impolite to show "6 PM" to each of them. They will incorrectly assume it is their local time and this could cause them to miss the deadline. FlyingFish is wise to this sort of thing and will use clues in the messages sent by the web browsers (the student machines) to try to guess the time difference involved, and then display times expressed in the local (student) time zone.

The GMT Offset value should be the number of seconds your time zone is offset from the time in Greenwich, UK – the time that used to be called "GMT" but is now called "UTC" (Universal Coordinated Time). Positive values are for places to the West of Greenwich and negative values are for places to the East (according to longitude). This value could in theory be obtained from the Date & Time control panel, but this separate control in the ini file has some advantages. For example, if you are hosting services on behalf of a university in another time zone, you might choose to make the GMT Offset match your client's time rather than your own.

8.5 persistent

These keys enable or disable persistent HTTP connections (HTTP 1.0 standard only). They are one of only a very few parts of FlyingFish that allow you to make a distinction between clients (students) using Netscape™ and those using Internet Explorer™. By default persistent connections are enabled for both browser types, and you probably should just leave it that way. Persistent connections are much faster and there is essentially no penalty for using them.

8.6 license

FlyingFish is commercial software and therefore we need some way to enforce the payment of license fees. Unfortunately this has meant that we now have a fairly aggressive license check on startup. The license check works like this:

1 Three strings are read from the jellyfish.ini file: license/hostIP, license/software and license.expires.

2 The license/expires value is converted to a "seconds" value using the standard FlyingFish date conversion.

3 A license string is built up by concatenating the hostIP, software and seconds values, e.g. 130.95.52.241-FlyingFish-102365774

4 The license string is then encrypted to make a 12-character license key e.g. 1anz-a8rq-9y4b

5 If the license key matches the jellyfish.ini file value "license", the license is considered valid and FlyingFish starts normally. If, however, there is any mismatch or missing value, FlyingFish will start but will set an internal timer that will disable operation after two hours.

When you first receive FlyingFish (by CD or from the download site) it will have a license key that was valid on one of our hosts. This key will no longer be valid on your host because your computer has a different IP address from ours. Therefore, if you are ready to do real serving work with FlyingFish, you will need to obtain a fresh license key. The status of your license key, and the procedure for acquiring a new one, are shown if you log in and then go to (e.g.)

```
http://localhost:8080/jellyfish/admin/license/
```

Because each of the license key subkeys hostIP, software and expires contribute to the license string, and the license string produces a unique license key, you can't modify any of the strings without invalidating your license.

The process of applying for a new license is somewhat automated but at some point you will receive your new license key by email. The email message will actually contain a complete license-UDB entry of the following form:

```
license=1anz-a8rq-9y4b
license/expires=31 Dec 2001 11:59PM
license/hostIP=130.95.52.241
license/software=FlyingFish
```

We could have completely automated the process of getting a fresh license. However we chose not to do this. The main reason is that we consider it a fundamental security principal that the FlyingFish *never* modifies its own jellyfish.ini file. Therefore you will have to perform the following steps to install a fresh license:

1 Quit your FlyingFish if it is running.

2 Use a simple text editor like NotePad to edit the file jellyfish.ini[16]

---

16 The default settings for Windows™ are such that, if you double-click the jellyfish.ini file, it will probably open up in NotePad. If not, you will at least have the chance to select NotePad from a list of applications, and it will then be used to open ini files from then on.

3 Copy the four lines that start “license...” (like the example above) from the email message.

4 Paste the four lines into your jellyfish.ini file, replacing the lines that are already there.

5 Close the NotePad window and save changes.

6 Test your FlyingFish and check that the new license has been accepted.

Strictly speaking only the license=1anz-a8rq-9y4b needs to be replaced, provided the on-line license application has been used, but it does no harm to install all the fresh strings, since they make a matched set.

8. The Done mechanism

We have seen how FlyingFish knows what work the student must complete, and how access to nodes is granted. In this chapter we explore how FlyingFish knows when a student has completed some work.

There are two main kinds of nodes:

1 "Wrapper" nodes that contain one or more problems; and

2 Problem nodes.

A problem node is associated with a single task or activity we expect the student to complete. If FlyingFish receives authenticatable information that such a task has been completed, it simply marks the node as Done and then activates the Done mechanism for the parent or enclosing node.

You will recall that a student is required to complete everything that matches the info/Complete string at a given level. The Done mechanism enforces this. It searches the nodes identified in the info/Complete string in the order given. If every matching node is completed, then this node is also deemed complete and the Done mechanism is invoked for its parent or enclosing node. This recursive process continues until we reach the top of the task tree (the empty branch). If this happens then the student is deemed to have finished the entire course and a message to that effect is generated.

More usually, however, some un-Done node is found and, because the search order matches the info/Complete order, this un-Done node is something the student might very well go on with. The branch of the un-Done node is saved in the student's user UDB file under the key "Branch/suggested" so that at any future time it can be returned quickly without an exhaustive search.

The detail of how FlyingFish knows that a particular problem node is Done depends on the problem type for that node. The approach for the main classes of problem are explained separately below.

8.1 An applet-question is Done

The security surrounding the java-applet question type is reasonably high. To keep it that way I don't intend to discuss it in too much detail here! Suffice it to say that we consider it quite unlikely that students will be able to pirate the applet-question-done message successfully. In other words, if FlyingFish receives a message stating that a certain applet has been successfully completed, and all security checks have been cleared, it is reasonable to assume that the student really has solved the problem. So the Done mechanism is simply invoked without question.

8.2 A Multi-choice question is Done

An MCQ-type question node contains a UDB file called "mcq" that contains all the information needed to both display and assess the question. FlyingFish receives the student's reponse as an HTML form-submission and compares it to the contents of the mcq file. If the question is correct, the Done mechanism is invoked. Note that FlyingFish security restrictions prevent the student from viewing the mcq file or any other UDB.

Note that the different question types do require some custom code at the FlyingFish server. It has to be able to take the different input forms and translate them into calls to the Done mechanism, or into messages for the student to

learn from. As new question types are developed, new server-side processing routines will have to be developed to handle the input, i.e. to decide when a problem has been completed. If you have a new problem type that you would like to implement, discuss it with our technical staff.

9. On-line adminstrative tools

9.1 Edit this file

Suppose you are in the computer lab during a class, and a student points out an error in your HTML solution for a problem. Normally you would have to make a note about it and fix it later. Meanwhile, 100 other students are probably viewing the same error and are quietly cursing the course coordinator (or else they are being misled by the error, which is worse).

FlyingFish provides a crude on-line editor for HTML so that users of sufficient authority can make instant changes. The standard FlyingFish release is set up to allow this on every page. Log in as a high-level user (e.g. a Developer) and look at the bottom of the page for a small link "edit this file". Click on the link. You should see something like this:

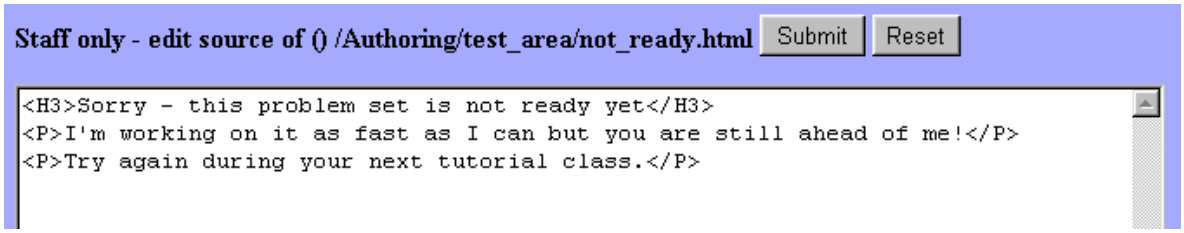


FIGURE 20 The on-line HTML editor.

Edit the text and then press the Submit button. You should see your changes immediately.

It is possible to edit any text-based file you can see e.g.

- UDB file contents
- HTML files
- text files

The editor is invoked using the query string

```
?Cmd=SOURCE&Pattern=t
```

and you will see that the "edit this file" link is nothing more than the line shown above i.e. it adds this "edit" query string to whatever the current Branch is.

By altering the Pattern you can edit files in the user, year or task tree. If no pattern is given, the default is 't' i.e. 'edit file at this Branch in the task tree'.

One of the great advantages of the on-line editor is that the internal caches for files are bypassed. In other words, if you edit some HTML or UDB file, the changes will be respected and will immediately be available to all users. If you edit files from the Windows® Explorer™, there is no such guarantee.

Warning – watch out for the </TEXTAREA> tag

The crude on-line editor is actually an HTML form and you can see it if you examine the contents of the file [task root]/edit_wrapper.html. FlyingFish puts the raw text of the edited page into the <TEXTAREA ...></TEXTAREA> tag pair in the form, like this:

```
<FORM METHOD=POST>
```

```
<INPUT NAME="Cmd" TYPE="hidden" VALUE="EDIT">
<TEXTAREA NAME="HTML" WRAP=TRUE ROWS=18 COLS=80>@{this}</TEXTAREA>
...
</FORM>
```

@this is a special HTML tag which means “the raw text of the file at the current Branch and using the current Pattern”. Unfortunately the browsers are currently not smart enough to deal with a tag combination like this:

```
<TEXTAREA>
Blah blah blah <TEXTAREA>Some text in my private textarea</TEXTAREA>
the browser will think that the edit_wrapper text area has finished and this text will appear OUTSIDE the edit box!
</TEXTAREA>
```

The text in italics is the raw text of a file that is being edited. But the edited file contains a <TEXTAREA> pair and this confuses the browser. The designers of the web browsers did not think of the possibility that someone would want to edit text containing the </TEXTAREA> tag in a <TEXTAREA>, so the <TEXTAREA> is terminated by the FIRST </TEXTAREA> tag that is encountered. This can cause some very strange effects. For the time being you should avoid editing files that contain the </TEXTAREA> tag, because this could lead to corruption of the file[17].

9.2 Directory

There are times when it is helpful to provide users with a dynamic listing of all files at a particular Branch. FlyingFish provides a convenient way to do this: the directory() function.

Log in as a Developer and then click the small link “Directory” at the bottom of any page. You will see something like Figure 21.

Figure 21 A typical directory listing

The browser is actually showing the processed form of a file at the top level of the task tree called “directory.html”:

```
<TABLE>
<TR><TD ALIGN=TOP>
<H3>task</H3>
@(t){directory()}
</TD><TD ALIGN=TOP>
<H3>year</H3>
@(y){directory()}
</TD><TD ALIGN=TOP>
```

17 The obvious work-around is for the FlyingFish to mangle the </TEXTAREA> tag into something else, like </_TEXTAREA>, when it is found in a file being edited. But this makes me uncomfortable because it is confusing to the user to see arbitrary changes to files. This may be an option in a future release. The other obvious work-around is to have an HTML editor (e.g. in Java) which is not itself HTML, again something for a future release.

```
<H3>user</H3>
@(u){directory()}
</TD></TR>
</TABLE>

["@(u){/user/Authority}"=="Developer"?@(t){/file_upload.html}]
```

Which in turn is mainly showing the results of function tags like

```
@(t){directory()}
```

i.e. “create a formatted HTML list of all files and folders in the task tree at this level”.

The directory() function is mainly to aid *staff* navigation, but it is available, in a limited form, for use by students. Staff see all files and folders; students see only the folders. This is not to improve security – because FlyingFish does not rely on “security by obscurity” – but so that a list shown to students will not be cluttered with other things. In addition, Student-level users cannot see any directory listing from any tree *except* the task tree.

Suppose you are setting up a course in Origami. You would normally provide links to each of the course topics on the page

```
[task  root]/Origami_110/body.html
```

something like this:

```
<A HREF=“paper/”>Where to buy origami paper</A><BR>
<A HREF=“swan/”>Exercise 1\: the Swan</A><BR>
<A HREF=“cat/”>Exercise 2\: the Cat</A><BR>
```

Where paper, swan and cat are all subdirectories i.e. subtasks of the Origami directory[18].

If you were feeling lazy, or if the course contents changed dramatically every few days, you might choose to provide a single directory() function tag instead of the three anchors:

```
@(t){directory()}
```

This would produce a list of the subdirectories for the student, complete with anchors to allow navigation. The only drawback is that the names of the directories will be in alphabetical order rather than the info/Complete order that you would probably prefer.

The directory() function is thus of only limited use for display to students. It might be helpful if you want to provide an alphabetical index or glossary at some point in your course material.

---

18 This is an example where some serious innovation will be needed, if the computer system is to mark the students’ work!

9.3 Class list

Log in as a Developer, then click the small link “Class list” at the bottom of the front page. You should see something like Figure 21.

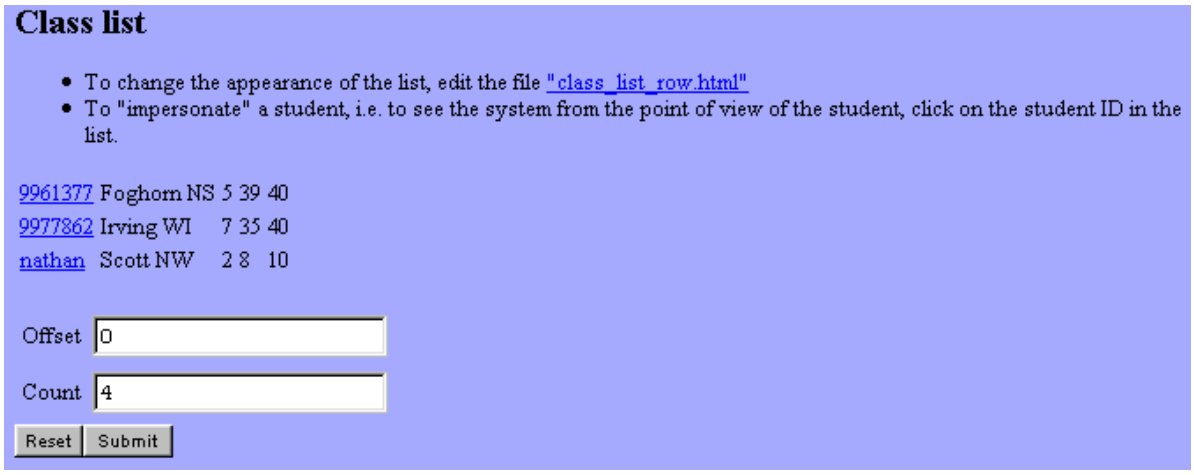


FIGURE 21 A typical class list

The list may not show all the users known to the FlyingFish (i.e. all the users in the user tree). The contents of the list are governed by

- The contents of the info/Student filter[19] at this level (or the first filter found while searching upward);
- Your user preferences for Offset and Count.

The offset and count values govern how many students will be shown at once. This feature is necessary because it can take a long time to display a list of several hundred students.

Each row of data in the class list is generated by processing the file class_list_row.html in the context of one user. In this example, the class_list_row.html file contains

```
<TR>
<TD><A  HREF="@(t){/cgipath.html}/admin/">@(u){/user/UserID}</A></TD>
<TD>@(u){/user/Name/Family}  @(u){/user/Name/Initials}</TD>
<TD>@(u){Done/a/ATTE}</TD>
<TD>@(u){Done/a/AVTM}</TD>
<TD>@(u){Done/a/MXTM}</TD>
</TR>
```

It will be seen that the columns of the table are

- a link to the user’s admin area i.e. the name and password change page
- a full surname-initial identifier
- the number of attempts at assessed problems at this level or below
- the total mark awarded for assessed problems at this level or below

---

19 This is explained in another section; use the table of contents and look in the chapter “Special Files”.

- the maximum possible mark for assessed problem completed at this level or below.

It is possible to customise the class list in a given course: copy and then edit the file class_list_row.html. This file is found by “climbing search”, so a file at a lower level will *override* one at a higher level.

9. The Forum

It is increasingly common for students to have access to computers outside of scheduled class times, e.g. at 2am from home. If a student needs help at such a time, what resources are available? Email, most likely, unless the student does not have an account. But then you must ask yourself, do I really want to receive 100 email questions about a particular tutorial problem?

We have found that a simple bulletin-board system can be tremendously helpful in teaching, and FlyingFish makes it easy to set up and use such a system. In fact, you might choose to use FlyingFish *mainly* as bulletin-board support for some other form of teaching i.e. you might not have any computer-marked tutorial problems at all.

Here we are using "bulletin board" to mean an *accumulation of messages*, rather like the bulletin board in a shopping centre, only on computer. The idea is that anyone can add a message to the bulletin board, and all messages are seen by anyone who looks at it. This is not the same as email communcation, where messages are *sent* from one person to another, and are not seen by anyone else. The great advantage of an *accumulation* of messages is that, once a particular matter has been discussed, it remains as a record for subsequent users. I'll spell it out: **you only need to answer a given question ONCE**. This can represent a huge saving of time for staff.

For historical reasons, the bulleting-board features of FlyingFish are called "the Forum"[20]. The bulletin board associated with a given part of the task tree is called "the Forum for ...". For the pedants among you, the plural of "Forum" is "Fora", not "Forums", but we are Engineers so we will accept either.

We have found that the keys to running a succesful Forum are

- They must be about *very narrow topics*. It is no good having a small number of general purpose Fora – students are forever short of time and they want well-targeted help.
- Staff must monitor Forum contents and respond quickly – otherwise the Fora will fall into disuse and may even be abused by some students.

9.1 Adding a message to a Forum

The Forum is all wired up and ready to go in the standard FlyingFish release. This is because Fora are created as they are needed, so you don't have to create anything up front.

Log in as a Developer and go to any page. Look in the top right corner of the header for a link "Forum". Click on the link. You should see something like this:

---

20 The first version of the Forum was inspired by commercial software called Pacer Forum™, available for Macintosh only in 1994.

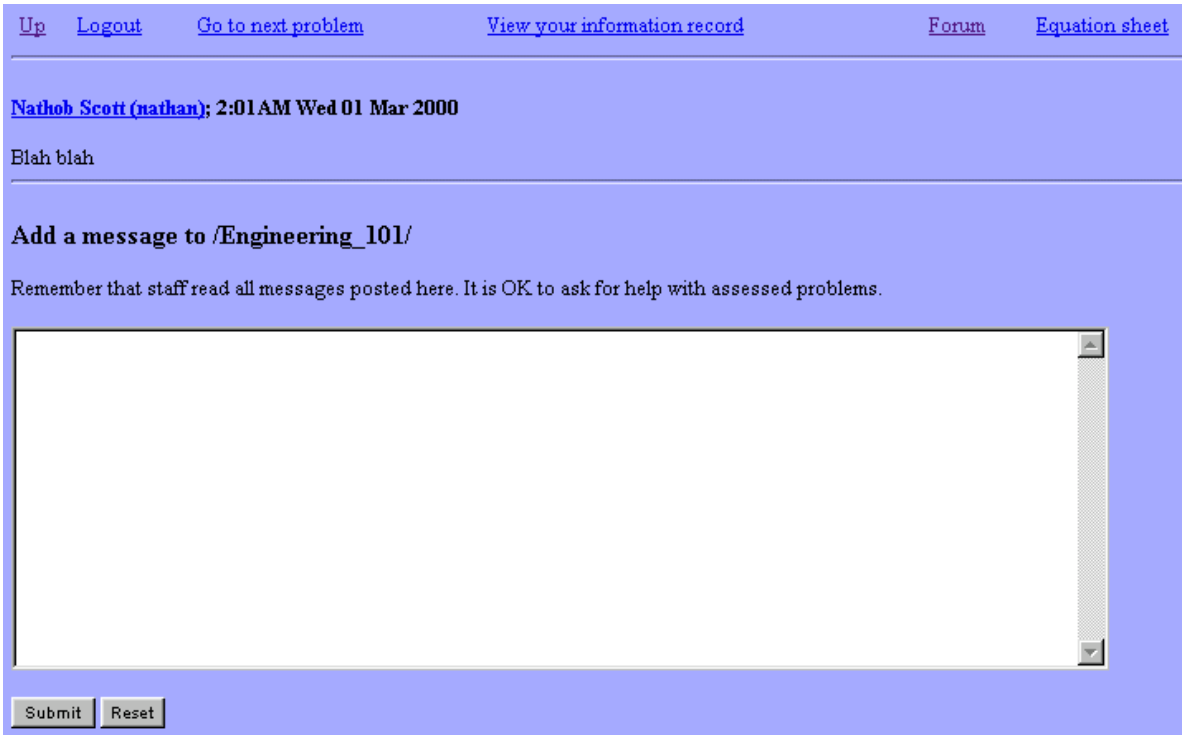


FIGURE 22 An example Forum.

If no-one has used the Forum for that Branch, then you will see only a message inviting you to add something. However, if other users have already contributed messages, you will see a list of these with names, times and dates, followed by a place to add a fresh one. Try typing something and then press the Submit button. You should see your new message at the end of all the previous ones.

Now use the "Up" or "Front Page" link to go to a different Branch. For example you might navigate downward into a course. Click the Forum link at this new Branch. You will see a different Forum from the one you saw before. That is because this is the now the Forum for the new Branch, and it is maintained separately from the Fora for all other branches. There is one Forum at each and every Branch in your course material.

The association of Branches with Fora should give you a hint about how they can be used. As we go downward into a course, the Branch gets longer, and the course material gets *more specific*. A top-level course such as "Dancing" might have a sub-directory (read sub*task*) called "Ballroom_Dancing", which in turn might have a subtask "Tango" and so on. Messages about the Tango should go in the Forum at the Dancing/Ballroom_Dancing/Tango branch. Messages about Dancing in general should go in the top-level Dancing Forum. We have found that students need a little guidance about this at the start of a course, but they quickly catch on.

Sometimes you might want to have several Fora at a single Branch, just as you might have several HTML documents in a single level of the task tree. This is no problem at all: *every HTML document* in the task tree effectively has a Forum associated with it, and these are separate from the Fora for the body.html documents.

9.3 Anonymity and carelessness

If you log in as a Student-level user, the Fora will look quite different. The names of most of the other system users will be hidden, and you will see only "Anonymous". This anonymity has proven very helpful in our teaching because we find that students are more willing to ask "dumb" questions if they have no fear of ridicule from fellow students. However it has the disadvantage that some students, failing to read the clear warnings that "staff read all messages", think that they are *completely* anonymous and can therefore write any sort of rubbish. Ah, youth. A few stern replies soon sort these characters out.

Students see the names of staff members, and they see their own name. All other user names are masked. Note that students do NOT see links to visit the admin areas of staff, and cannot visit these even if they edit the location string[21].

FlyingFish is designed for use by students and includes features to correct errors when they occur. Students and staff can use HTML tags in their messages BUT what would happen if the following message were posted?

```
<H1>Somebody please help me,
I've missed all the lectures
```

The user has opened the <H1> HTML tag but has failed to close it. Normally this would be a disaster for the remainder of the Forum because all subsequent text (until the first </H1> tag, if any) would be in huge type. Actually this might not happen but the results of improperly nested tags are undefined in HTML so the result could well be quite strange. However FlyingFish is smarter than that and corrects the HTML syntax of messages posted to Fora.

Note that the example message includes a carriage return, and that this would normally be ignored in HTML i.e. the resulting message, as rendered by the browser, would all be on one line. As a concession to users who do not necessarily know HTML, FlyingFish replaces carriage returns with
 tags, so that the message looks OK when it is displayed by the browser.

The Syntax correcting mechanism is only invoked when a message is posted to a Forum by the normal means; it is NOT invoked if you edit the file using the "Edit this file" link. FlyingFish assumes that a staff-level user knows HTML and will allow you make a mess of it if you want to.

9.2 The Forum change list

When a user adds a message to a Forum, how do we find out about it? At the moment the only way to see what has happened in the Forum is to visit a special page that shows Forum changes. Go to the Front page and then select the link "Recently changed forums". You should see something like this:

---

21 Because a Student-level user cannot impersonate *any* other user, let alone a staff user. If they edit the ~userID part of the URL they will simply be shown their *own* admin page.

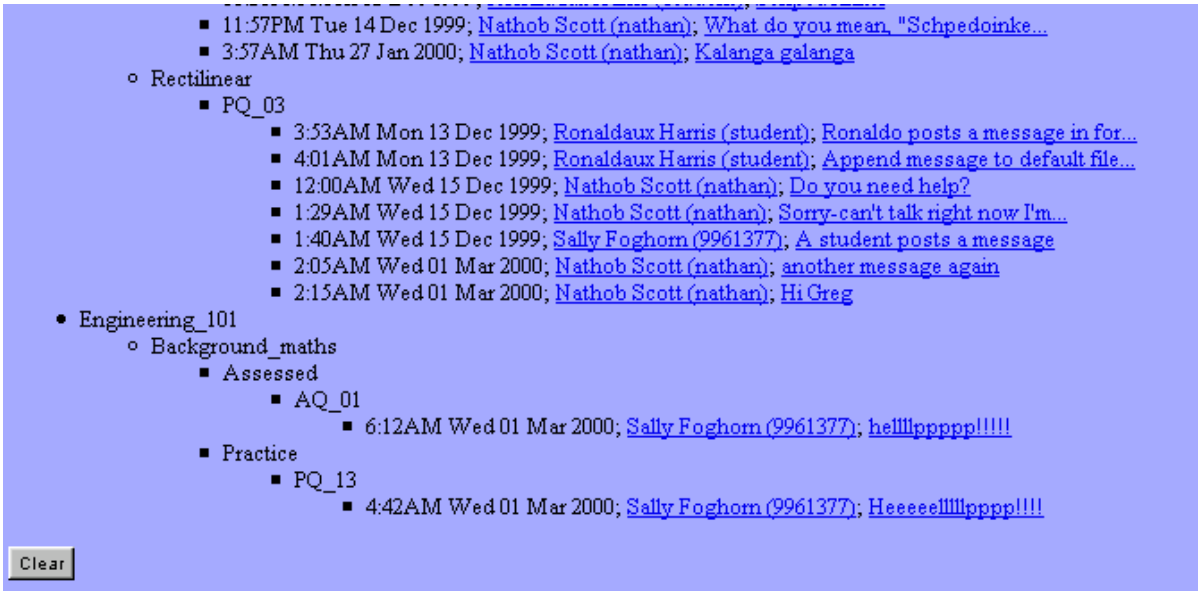


FIGURE 23 Recently changed forums

It's a *list of lists* and the nested structure reflects the structure of the task tree. If messages have been posted, there are times and dates, a user name link, and a synopsis of the message. If you click on the user name link, you will see the admin area for the user who posted the message i.e. the page showing the names of the user. If you click on the synopsis of the message, you will go immediately to the message where it was originally posted. If you click the Clear button, then the list will be emptied, but this only affects your change list, not the change lists for other users.

Forum change notifications are stored in a UDB file in each user's directory e.g.

```
[user  root]/nathan/forum_changes
```

this UDB grows as messages are posted, and when you press the Clear button it is erased. We want Student level users to be able to clear the list so we specifically provide change permission for this file (see [task root]/forum_changes).

9.3 Editing Fora

If you need to edit a Forum, click the "Edit this file" link as usual. You will see a series of messages in the following format:

```
<A NAME=951919921></A>
<H4>@(t){publicID(9961377)};   @(t){date(951919921)}</H4>
hellllppppp!!!!!
<HR>
```

If you want to delete a message entirely, delete everything from <A NAME=... to <HR>, a complete repeating unit.

The line <A NAME=... is the anchor used to scroll the Forum to the message. It is not critically important but you should leave it alone if you want the forum_change anchors to work as users will expect.

The line <H4>@(t){publicID(9961377)}; @(t){date(951919921)}</H4> is processed to become the name, date, time string associated with the Forum message. publicID() turns a userID into something suitable for viewing by the current *actual* user. date() displays the time of the message correctly, regardless of the time zone of the user or server. It is probably a good practice not to edit this line in a Forum, unless you really know what you are doing. It is OK to delete it if you plan to delete the whole message but be aware that users could then try to jump to a non-existant message (in this case they would see the first message in the Forum because the <A NAME=... tag would not be found).

9.5 Behind the scenes: where is the Forum data?

Forum files are stored in the year tree, along with other "ephemeral" data such as user count statistics and system log files. The idea here is that the task tree contains data that is inherent in a course or set of courses and which *will not change from year to year*. However the year tree stores anything which will probably be refreshed each year.

We have the practice of renewing the year tree each year (or semester). It's easy to do: we simply change the jellyfish.ini file to show a new Path e.g.

```
tree/y=../2001b
```

The old year tree need not be deleted – it contains important historical data and Forum messages which can be used to refine your teaching. However the new tree/y path identifies a new tree so students have the chance to post messages into clean Fora, without seeing the messages of the previous semester.

The year tree has a directory structure which is *parallel to* the task tree, and the Fora are stored with the same file names as their task-tree shadows. The main Forum for a particular branch, for example, is a file called body.html somewhere in the *year* tree.

9. Introduction to Java and FlyingFish: the single-answer question type

9.1 Standalone Java

The applets that allow students to solve assessed problems on-line are *networked* i.e. they send and receive network messages. This can be contrasted with the more usual kind of Java applet that does not try to use the network and thus *stands alone* – it works without the FlyingFish server.

If you are interested in writing some new Java applets for a course, a good way to start is to get your applets working in Standalone form before you try to connect them to the server. Standalone applets are easier to understand and debug.

9.2 The Single-answer question standalone shell

Although in theory one can write an arbitrary Java applet and turn it into a working problem, there is also something to be said for recycling existing problem code. The key question is whether your educational needs can be satisfied by adapting one of the existing problem types, or whether you really do need something quite new.

In this section we will discuss the process of writing problems that look like Figure 16:

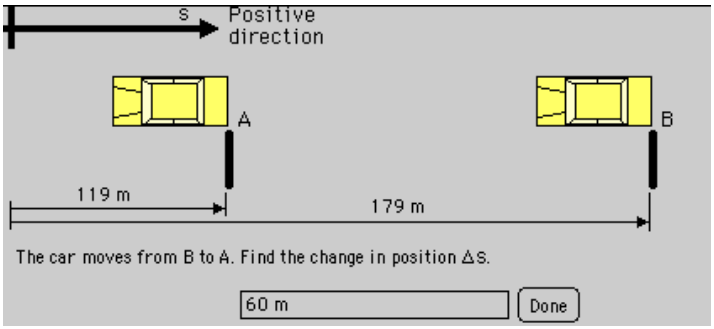


FIGURE 20 A screen shot of the applet Rect1010 in action.

This is NOT an arbitrary java applet - it is a special type of applet that knows how to

- draw a background picture
- choose and display numbers for some physical system
- calculate the expected answer and a range of expected incorrect answers;
- receive student input in the form of a single numerical answer with units
- provide feedback for the student answer.

This networked version of this problem is derived from a standard Jellyfish library class called CSingleAnswerQuestion, which in turn is derived from more abstract classes that know how to communicate with the server.

The single answer problem type is not very exciting to look at but

- it is very easy to write new problems of this kind, and
- if the problems are carefully designed, they can be very helpful for students

In other words, this apparently boring problem type can be used to set up a very efficient and helpful tutorial set. We have used problems of this type to teach students in Engineering Dynamics since 1995.

9.3 How to get started

In order to write single-answer questions you need to be comfortable compiling and running Java code BUT you do not have to know much Java. You will be amazed at how little of the language you need on order to write Engineering or Physics problems. You will be working from an existing code template and most of the new material you write will be numerical calculations - very like C or Fortran.

If you want to learn Java then I recommend The Java Tutorial by Campione & Walrath.

The main thing you need right now is a Java compiler. I use Metrowerks CodeWarrior, which is cool, inexpensive, and works on both Windows and Macintosh computers. Some of my colleagues seem to like Microsoft Visual J++. The only compiler I will warn you NOT to try is the free "Java Development Kit" or JDK which is available for download from Sun. It is tempting because it is free BUT it is command-line based and is completely unfriendly to a novice developer. In other words, you will be driven mad because you will not be able to control it or understand what it is doing. So spend a few hundred dollars and get a proper GUI development environment.

9.4 Setting up a project for the single-answer example

Here is a picture of my CodeWarrior project on a Win32 computer:

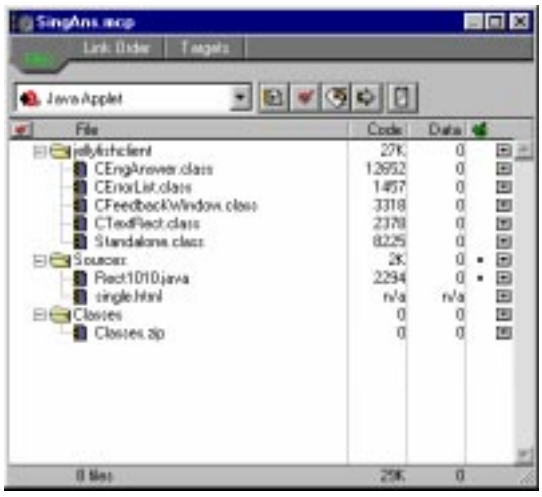


FIGURE 21 Source files to compile Rect1010.class

Note that the only actual Java source file is Rect1010.java. All the other files are compiled .class files or other things like HTML.

Here is a picture of the directory structure I work with on a Win32 computer:

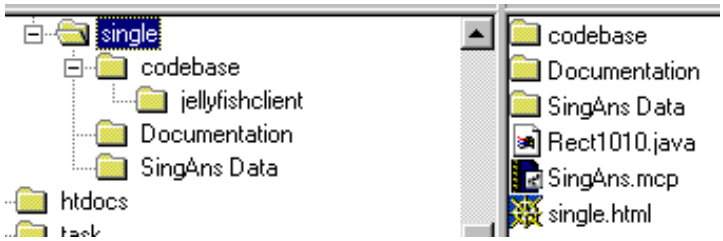


FIGURE 22 Arrangement of project and compiled files.

The important point here is that your project is supposed to produce a single output file called Rect1010.class, and put that file in a directory called codebase. The codebase directory already has a subdirectory called jellyfishclient which contains the compiled classes e.g. CEngAnswer.class. These can be seen as library files. Note that the codebase folder also contains the image for the problem (problem.gif) and also a little GIF containing a symbol used in the problem (DELTAS.GIF).

8.4 Understanding the single-answer-question code example

Now for the explanation. Here is the source file Rect1010.java, with some extra comments in italics:

```
import  java.applet.*;
import  java.util.*;
import  java.awt.*;
import  java.math.*;
import  java.net.*;
import  java.io.*;
import jellyfishclient.*; // base classes for all question types
Very like #include in C or C++
public class Rect1010 extends Standalone
This defines the new java class we are going to compile. Note that the source file name
MUST reflect the name of this class. So if you change the name (Rect1010), remember to
also rename the source file. Note that java is strongly case-sensitive.
{
    double carDistA, carDistB;
Define variables which will be used in both of the methods (procedures) in this class. These
are sometimes called "instance variables" of the class. The "double" type is just an 8 byte
real number.
public void ChooseVariables()
This is the method which chooses random values for the problem parameters, and also does
all the calculations.
{
    carDistA = RangeRandom(40, 120, 1);
    carDistB = carDistA + RangeRandom(40, 120, 1);
This is the standard way to choose random variables. The RangeRandom() function is
defined by the libraries. The arguments are (min, max, step) and the result is a double.

    ClearErrorList();
Tells the superclass to forget any previously loaded list elements.
    // remember that the first "error" is the correct answer.
    LoadError(carDistA - carDistB, "m", "", "", 0, 1, 0, 0);
```

The superclass keeps a list of numbers with some associated information. This line loads an element into that list. The first such element is taken to be the correct answer.

```
// ans, units, title, message, severity, c0, c1, c2 [competence classes]
LoadError(carDistB - carDistA, "", "sign.html", "You seem to have forgotten about the sign convention; this is far more important than you may realise.", 5, 1, 0, 0);
LoadError(-85.0, "", "yourNumbers.html", "Remember you have your own numbers for every problem.", 3, 2, 0, 0);
```

Here we load some erroneous answers. We try to pick things that students will probably get wrong, based on past examination papers. The -85.0 answer was the answer to a problem on a printed sheet that was given to the students.

```
}
```

public void paint(Graphics g)

paint() is a method defined by the Java class Applet. We "override" it to draw something that is specific to this problem.

```
{
```

"super" means the super-class of this class i.e. Standalone. That class has a paint() method which does nothing but draw a picture called "problem.gif", if it can be found.

```
// If your problem does not need a background picture, just comment out the next
//  line.
super.paint(g);

if (!done_loading_seed) return; // "numbers" have not arrived yet
// nns() is a utility defined in Standalone.class. It formats a double as a String in a
// sensible way i.e. it will never produce something awful like
//  "4.200000000000001".
g.drawString(nns(carDistA) + " m", 45, 115);
g.drawString(nns(carDistB) + " m", 220, 122);
```

When you first write a problem, give any old values for the positions (e.g. 45, 115). Then compile it and run it. When you click on the applet the coordinates of your click are shown in the console window and you can then set the coordinates here correctly. It is a little painful but you can get them right in one "iteration".

it is OK to put words in either the HTML file for the problem or the problem.gif image. This is a demonstration problem so this next bit is to show the technique for mixing text and small graphics e.g. Greek symbols.

```
// define some local integers to help position the text in neat rows
int xPos = 10, yPos = 152, dyPos = 20;
```

DrawString with a capital D is defined in Standalone. It is exactly like g.drawString but it returns the x coordinate of the end of the string, which we will need in a moment

```
xPos = DrawString(g, "The car moves from B to A. Find the change in position ", xPos, yPos);
// now we need to draw the symbol "delta S" at the end of the previous string
Image deltaS = getImage(getCodeBase(), "deltaS.gif");
```

note that images are drawn with the top left at the coordinates you specify, whereas text is drawn with the coordinate at the bottom left of the first character. So a correction is needed to place the image in the expected position on the line. You will have to determine this by trial and error for each little image you make...

```
g.drawImage(deltaS, xPos, yPos - 9, this);

// add a full stop after the image
xPos += deltaS.getWidth(this);
```

```
        g.drawString(".", xPos, yPos);

    one more trick you might need - drawing odd symbols like "degrees". This code is clunky
    but it works. A table of available characters can be seen at
    http://www.mech.uwa.edu.au/HowTo/JavaChars.html
        // Character degreeSym = new Character((char) 176);
        // g.drawString(nns(angle)+ degreeSym.toString(), 90, 50);
    } // end paint() method
    } // end class definition
```

9.5 Writing a new problem

1 Duplicate the file Rect1010.java under the operating system. Give the new file a name that describes the problem e.g. "Thermo21.java".

2 Import the new file into your project. You can leave the old file(s) in the project if you wish - they will not interfere with the new one. I have a project for one course with more than 200 applets in it!

3 Rename the existing problem.gif to (e.g.) Rect1010.gif so that it will not be replaced when you create new artwork for Thermo21.

4 Open the new file and change the name of the public class to match the file name (Thermo21 in this case).

5 Open the file Standalone.html and change the name of the class that appears in the <applet> tag (code="Thermo21.class")

6 Create artwork for the problem in a paint package and save it as problem.gif. Put this new GIF file in the codebase directory.

7 Edit the Thermo21.java file so that it calculates the answers you want, and displays them on the screen.

8 Compile the Thermo21 class file.

9 You may have to adjust the applet size shown in the HTML file i.e. you may have to edit the height and width values. If so, it may be a good practice to save the previous HTML file before modifing it for each new problem - this will save you the trouble of determining the new height and width again later on.

If your development environment does not allow you to "run" applets with debug support, you will need to make use of the *java console* to get information about what the code is actually doing. This means inserting lines of this kind:

```
System.out.println("*** carPosA = "+nns(carPosA));
```

This will make a string like "*** carPosA = 119" appear in the java console.

9.6 Importing the new problems into the Jellyfish tutorial environment

The "standalone" class is exactly that – it is not connected to the FlyingFish tutorial environment and only exists to make it easier to write and debug new problems. Once your new problem is working, you will want to install it as something that students will use. This means that the new problem will now be in contact with the server to get information about the student (such as the

random seed for the random problem parameters) and also to send information back (such as whether the student has had an attempt at the problem).

It is easy to switch a problem from a Standalone version to a networked version. You should create a new project file which compiles the networked versions because you will use a different set of libraries. Instead of the set shown above you will need to import all the classes from

c:\FlyingFish\htdocs\JellyfishClient\Questions3\jellyfishclient\

(of course, your actual Questions directory could live somewhere else - this is just an example)

Change the line

```
public class Thermo21 extends Standalone
```

to

```
public class Thermo21 extends CSingleAnswerQuestion
```

Compile your code.

Put the compiled class file Thermo21.class in

c:\FlyingFish\htdocs\JellyfishClient\Questions4\

Refer to the new class in a body.html document.

10. Writing your own compatible Java applet

It's possible to do great things with the humble Single-answer question type – as we have been demonstrating in Engineering subjects for years. However, if you have a vision of a new type of problem, you will need to create the interface in Java, and then wire it up to the FlyingFish server.

10.1 Write your own (incompatible) Java applet

Your work is to create something that will show the student something, ask a question, get some input, and assess whether it is correct or not. Easy! (No, definitely not easy, but worth doing anyway). The point here is that you should *thoroughly test your applet* while it is in a "stand-alone" form. It is still possible to debug an applet once it is "networked", but it is a little more difficult.

To make this discussion more concrete, let's use an example. Let's suppose you have developed an applet that has the student draw a vector on the screen. See Figure

10. Application example: the Java drag-and-drop question type

This chapter draws heavily on the work of Dr Moira Maley of the UWA Faculty of Medicine. In 1998 she started to use the FlyingFish to develop course material in the field of Microbiology. To satisfy the perceived educational needs of part of the course, Dr Scott and Dr Maley developed a new Java problem type (applet) for the system, which has proven to be both effective and adaptable.

The best way to get acquainted with the drag-and-drop applet is probably to try it "live" at

```
/Authoring/Micrographs/Breast2/
```

You will see something like this:

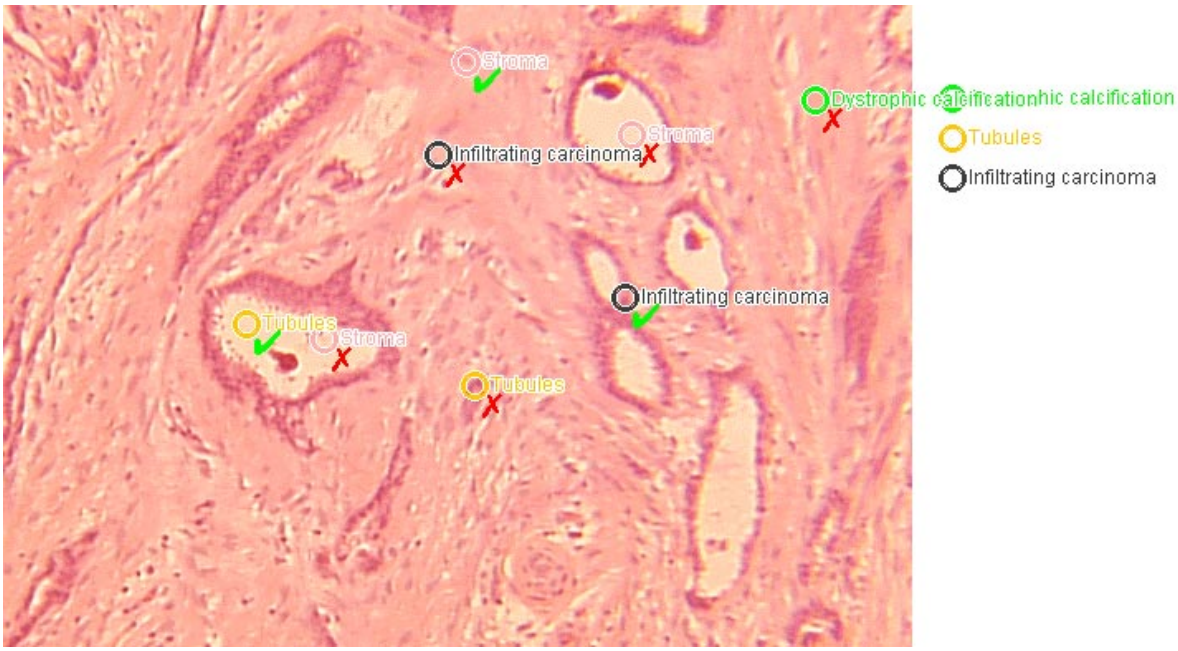


FIGURE 23 The CJavaMicrograph applet again.

This chapter explains how to create a new problem that has the drag-and-drop interface.

Before I launch into the explanation, I have some good news and some bad news. The good news is that you will *not need to know any Java* to create new problems of this type: you can simply re-use the existing applet. The bad news is that Dr Maley and I worked on this for several months and we are very proud of the Java programming that went into making it all work. This means that I will *not* give you the source code for CJavaMicrograph, so you won't be able to make changes to it. The next chapter contains some information about the approach to writing a new problem type, and you may be able to invent your own version of CJavaMicrograph – or something even better – based on that.

To create a new drag-and-drop problem, you need to create only three new files:

- The problem picture, called problem.gif or problem.jpg;
- An additional picture called the mask (mask.gif), and
- A text file called assessor.txt

As an example, in the following sections I will explain the steps to create the following drag-and-drop problem from the Engineering field of Statics:

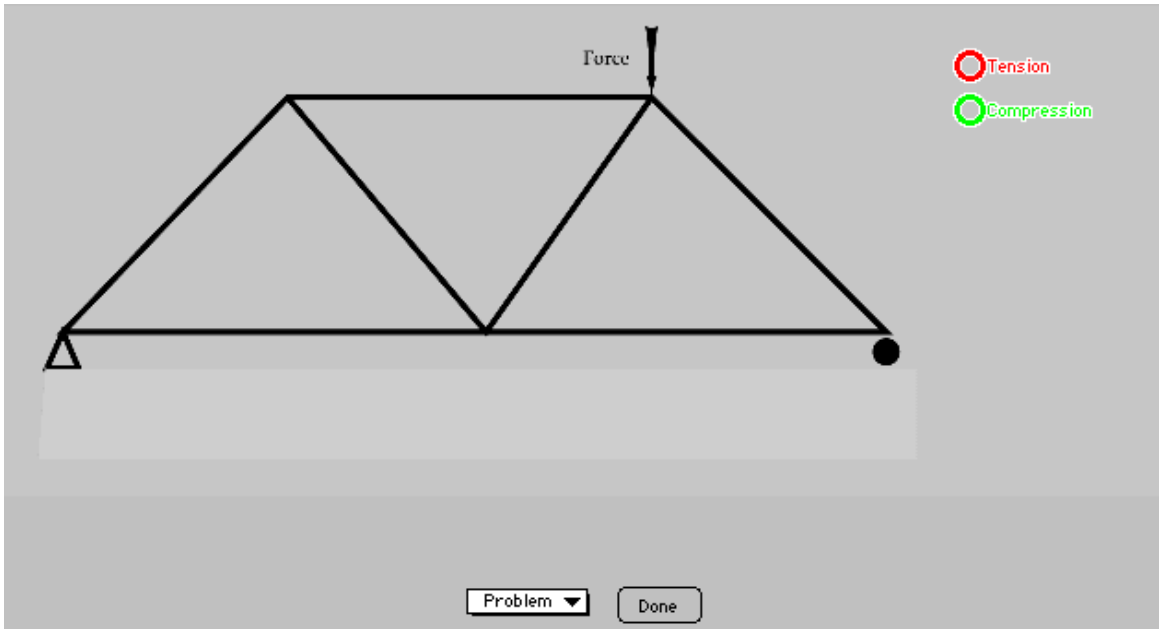


FIGURE 24 An example drag-and-drop problem from Statics.

10.1 The problem image

Can be anything you like. I created this artwork for the Statics problem using Adobe™ Illustrator®:

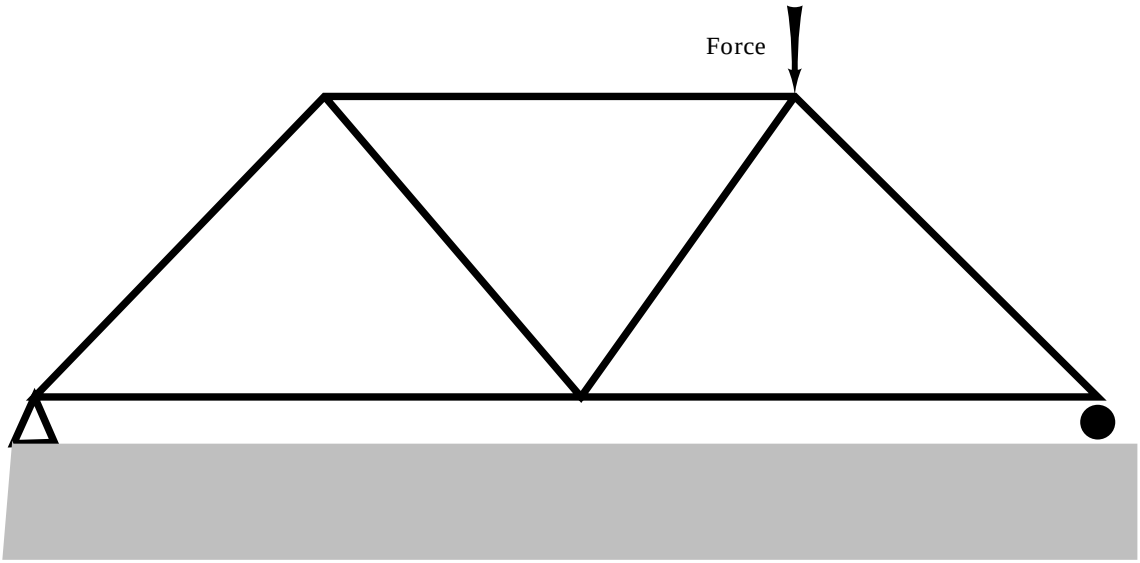


FIGURE 25 The problem picture for a problem in Statics.

The problem picture can be in either GIF or JPEG format. GIF is best for images with larges areas of solid colour (like Figure 26). JPEG works best for photographic images. Your picture must have the name problem.gif or problem.jpg. Note that the suffix is jpg and NOT jpeg.

10.2 Create the mask

The mask image must be the same pixel dimensions as the problem image. It is an image containing blobs or regions of solid colour that correspond to the semantically distinct regions in the problem image. This will be clear if I show you the mask for Figure 24:

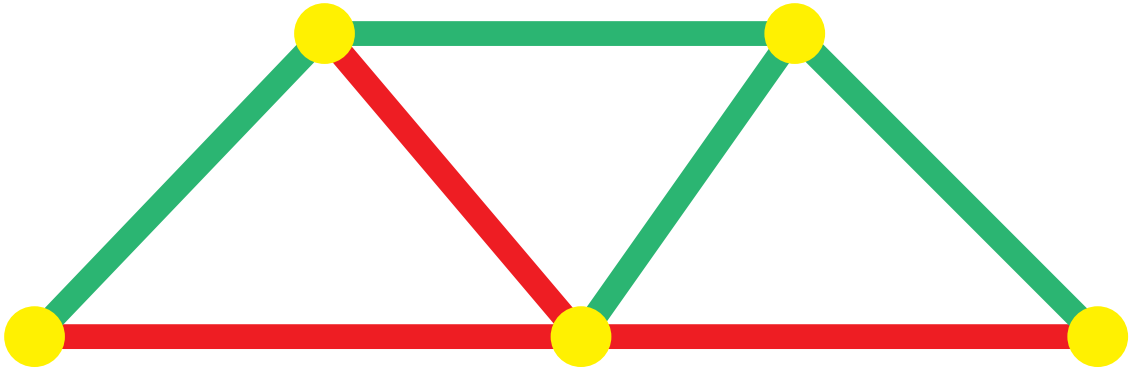


FIGURE 26 The mask image for the Statics problem of Figure 24.

You will notice that there is a fair amount of blank space around Figure 22. This is because the solid regions in the mask have to line up exactly with the problem image. In the photographic industry, two such aligned images are said to be *registered*. This will be clear if we show the mask image with the problem image superimposed, pixel for pixel:

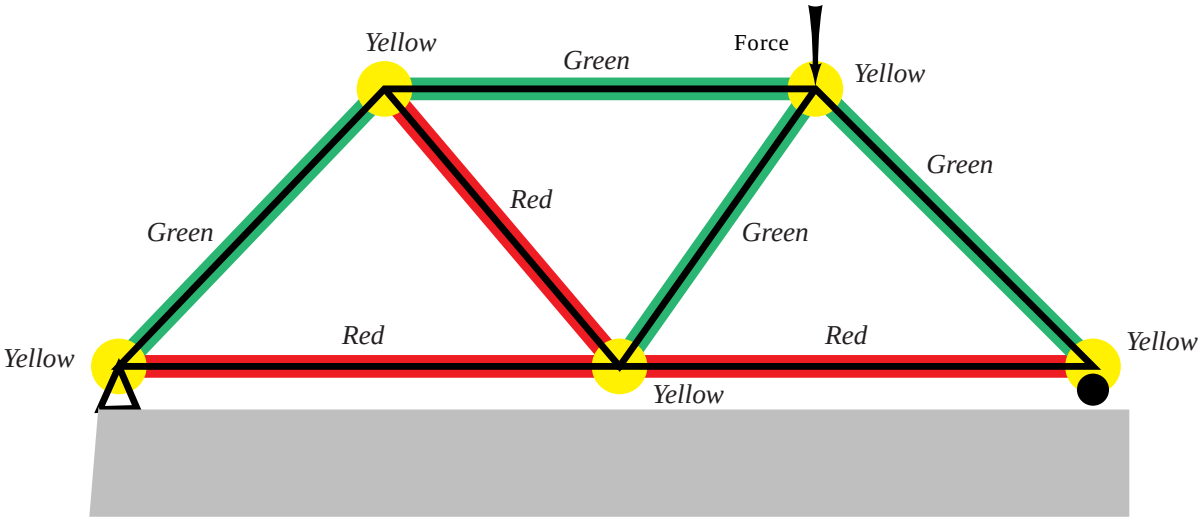


FIGURE 27 The mask image with the problem image superimposed. The colours of the semantic regions of the mask are shown in italics (since you are probably reading a grey-scale paper copy).

Note that the sausage-shaped mask regions are quite a lot bigger than the corresponding truss elements. We do not care whether the student puts a marker *exactly* on a truss element – near enough is good enough. Also note – and this may not be clear if you have a black-and-white copy of this document – that some of the regions in the mask image are solid red and some are solid green. The joints of the truss have solid yellow circles over them (explained below). The remainder of the image is solid white. There are no half-tones and the edges of the coloured regions are pixel-sharp and jagged. Why? Because the applet relies entirely on the RGB colour values it sees in the mask image, so if there are peculiar colours in the mask image, they can confuse the algorithm that tries to give feedback.

10.3 The assessor.txt file

By now you are probably familiar with UDB files that have no file extension. For historical reasons (because of errors I made early on), the assessor.txt file really is a UDB but it is stored with the .txt suffix. Sorry. Anyway, here is the assessor.txt file for the Statics example:

```
// This is a UDB file to assess a micrograph

// We start by defining the labels we want the student to position
Labels=Tension, Compression

// Define colours used in the masks
Color/255000000=tension
Color/255255000=joint
Color/000170085=compression
Color/000000255=right_member
Color/255000255=central

// Now we give details about each of the Labels defined above
Tension=7
Tension/min=3
Tension/radius=40
Tension/color=red
Tension/tension=Correct.
Tension/joint=Too close to joint.\nPlease try again.
Tension/compression=This member will be in compression.
Tension/right_member=This member must balance the vertical\ncomponent of the reaction\nfrom the roller support.
Tension/central=This marker is within the truss but it should\nbe placed on one of the structural elements.
Tension/baffled=This marker should be placed somewhere on the truss.

Compression=7
Compression/min=4
Compression/radius=40
Compression/color=green
Compression/compression=Correct.
Compression/joint=Too close to joint.\nPlease try again.
Compression/tension=This member will be in tension.
```

```
Compression/right_member=Correct.
Compression/central=This marker is within the truss but it should\nbe placed on one of the structural elements.
Compression/baffled=This marker should be placed somewhere on the truss.
```

This UDB file is also unusual in that it has C++-style comments, i.e. // to start a comment. I must emphasise that comments of this kind are NOT a standard UDB feature. I broke the rules for assessor.txt files because I felt comments were essential to help keep track of all the keys in the file. The file breaks one final UDB convention because the *order* of lines in the file is somewhat important, as we shall see.

The comments in the assessor.txt file are almost enough to explain what it does, however:

- There is a comma-delimited list of the Labels (doughnut names). These Labels can contain spaces if you wish.
- There is a block of definitions of mask-colour names e.g. Color/255000000=tension. The convention is a little strange and I apologise. The colour is given as a Red, Green, Blue triple (as in HTML background colours) but the three components are packed together as *decimal* numbers rather than *hexadecimal*. I no longer remember why I did it this way. Anyway "255000000" means "red=255, green = 0, blue = 0" i.e. primary red.
- For each doughut Label there is a single UDB key at the top level of the file. However the key can have many sub-keys, which are explained below, and which inform the behaviour of the doughnut being defined.

Note carefully: there are *doughnut labels* and there are *mask-colour names*. They can be the same if you wish but need not be.

Now let's examine the detailed information pertaining to doughnuts. Consider doughnuts with the label "Tension":

key meaning

Tension=7 There will be 7 doughuts with the label "Tension", initially all on top of each other at the right edge of the applet.

Tension/min=3 We will require the user to get three doughuts of this kind in the right place.

Tension/radius=40 The user is not allowed to position two of these doughnuts closer than this radius from one another.

Tension/color=red This is the colour the doughnuts will have.

Now we are getting into the assessment part. This is where the order of lines in the pseudo-UDB file starts to be important. What follows is a series of tests that we will perform on a doughnut to classify where it is on the mask, i.e. what semantic region (colour) it is over. The order of tests is important because the CJavaMicrograph applet actually supports multiple masks (something which is not explained here – too complicated and this is an introductory exercise). Each diagnostic test is of the form

```
Doughnut label/mask-colour name=feedback message
```

Again, be careful to distinguish between doughut labels and mask-colour names, particularly if some of your labels and colour names are the same. The meaning of the above is “If the user puts the doughnut with this label on the mask colour with this name, we will give them the following message”.

Generally the feedback message string can be anything you like. However it should be kept short because this is the string that will appear in the little “pop-up” when the mouse is over an incorrect doughnut. If there were too many words in the message, they would not all fit on screen and the meaning would be lost. Note that you can break a feedback message across several lines (i.e. you can put a carriage return in the message) using the escape sequence \n. Those of you who program in C, C++ or Java will recognise this as the newline character escape sequence.

There is one special feedback string: Correct. If you use this string you are telling the applet that the combination of label and mask colour is correct. You can have additional words after Correct if you wish, e.g. “Correct. What a legend.” and it will still be treated as Correct. You can also mark several label/name combinations as Correct if you wish and all will be accepted.

Note that the label/colour name pairs form a kind of look-up table spanning all the possible combinations. Note that we have not defined the mask colour “white” and yet most of the mask image is white. If the user places a doughnut on the white region, all the diagnostic tests fail. The line

```
Tension/baffled=This marker should be placed somewhere on the truss.
```

becomes active. The colour name “baffled” is reserved and means “any RGB colour not mentioned in the other diagnostic tests”.

10.4 Organisation of problem files.

If you follow the example CJavaMicrograph problems in the standard FlyingFish installation, you can’t go far wrong. The arrangement of the files is *not* arbitrary and the applet *will* look for exact file names in exact relative positions. Specifically the applet expects a directory arrangement like this:

- *problem directory*
  - body.html containing applet tag
  - problem.gif (or, if you wish, problem.jpg)
  - *mask0 directory*
    - mask.gif
    - [info file to protect mask.gif image from prying eyes?]

It is also possible, but not essential, to have a solution directory.

11. Glossary

Branch A path from the top of a tree structure to a particular node. The first part of a Branch corresponds to a directory/subdirectory path in a computer file system; the latter part can refer to sub-UDB entries in a single file. See also UDB. The Branch uniquely identifies the part of the problem set the student is currently working in. It can be inserted into a page using the variable notation @(t){/branch}.

Node Part of a tree structure which can be identified by a unique Branch string.

Problem or Question A single activity which a student-user is to interact with. A problem usually appears on a single Web page.

Problem set A group of related problems. The usual relationship is that the problems are about the same branch of knowledge, and they are arranged in order of difficulty. Problems in a set can be of any type and do not have to be all of the same type.

Problem type A class of problems that share a similar interface, that is, they look similar to the student. The interface consists of two parts: the part seen by the student, and the response expected from the student. An example is a problem which presents a diagram with some random parameters, and requires the student to type a numerical answer. All problems that provide these interface features are said to be of the same type. Typically problems of the same type have some common code.

FlyingFish An abbreviation for “FlyingFish Tutorial Environment”

FlyingFish A server program for Windows NT™.

Jellyfish Tutorial Environment The collection of platform-independant standards for directory trees and file formats, along with one or more executables for a specific computer platform, that together make the tutorial system described in this manual.

Task node See Node.

Task tree The collection of directories and files that describes all courses on a particular server. Each course appears as a subdirectory of the top task node; each course usually has several subdirectories representing topics; and each topic usually has several subdirectories representing problems. The tree structure can be as deep as is needed, provided it does not exceed the specifications of your computer’s file system.

Top task The highest-level task node. This node corresponds to the task directory, which on some systems is nominated in an initialisation file such as “FlyingFish.ini”.

UDB Universal Data Base: a file format developed by Dr Kevin Judd. It is text-only and usually human-readable. It is a list of key=value pairs separated by return or newline characters. The keys can contain heirarchical divisions, for example aa/bb/cc=value. The UDB file exists at a particular Node (i.e. in a particular directory) so it is possible to refer to any value in the file using an extended path, e.g.

topNode/sub1/sub2/udbFile/aa/bb/cc

Identifies the value with key aa/bb/cc in the file udbFile which is in the subdirectory sub2.

12. Appendix 1 Staying in touch

12.1 email addresses

Please keep us informed about what you are doing. We are always delighted to hear from people who are using our software, and we always answer email:

Dr Nathan Scott	nscott@mech.uwa.edu.au
Dr Kevin Judd	kevin@maths.uwa.edu.au
Prof. Brian Stone	bjs@mech.uwa.edu.au

12.2 Web addresses

Advance Education:	http://ae.maths.uwa.edu.au/
FlyingFish	http://www.mech.uwa.edu.au/nws/ae/

12.3 FTP server

Download the latest binaries and course material. Use a proper FTP client like WS-FTP because you must enter a user name (aeclient) and password (ask for this):

FTP server:	www.mech.uwa.edu.au
directory:	ae

13. Appendix 2 Frequently Asked Questions

Do I have to use a Windows™ server?

The first version of this system was used by students at UWA in 1995. In 1995 and 1996 the server was a Macintosh. We still love Macintoshes and do all our best work on them, but the fact of the matter is that most academics have access to at least one Windows computer but there are plenty who have no access to a Macintosh. So we gritted our teeth and re-wrote the server for Win32. The answer to your question is, yes, you do have to have at least one Windows computer in your life in order to use this version of FlyingFish. But try to keep this in perspective: every other computer in your world can be something else (Mac, UNIX, whatever) as long as it can run a reasonable Web Browser. Is it really so bad having just one Windows computer around? Also, have a little faith: we will eventually release a fully compatible macintosh FlyingFish... and all your course material will continue to work with it...

14. Appendix 3 How to find your own IP address

If you are only planning to log in on the server computer i.e. if your web browser is running on the same computer as your FlyingFish server, then you do not even need to know what your IP address is. Your web browser will understand that if you type

```
http://localhost:8080/
```

or

```
http://127.0.0.1:8080/
```

that you want to see the index.html page on your local server.

However, if you want to log in from any other computer, you will need the IP address of the server computer.

Windows® 95 and 98

Use the Start menu on the server, select the Programs sub-menu and then the MS-DOS prompt command to create an MS-DOS window. Type the command

```
winipcfg
```

And press return. You should see a new window like this:

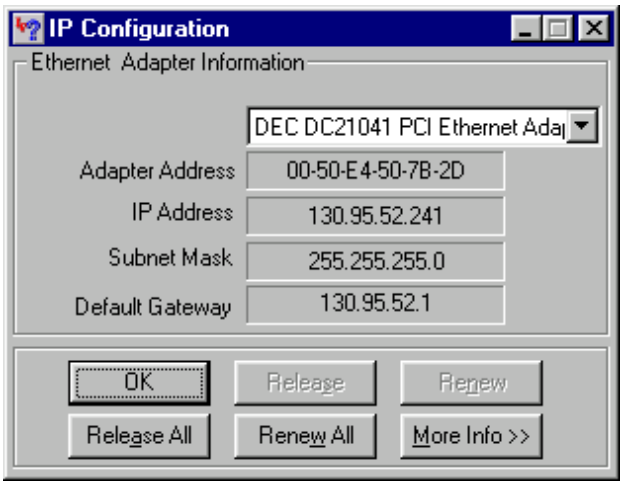


FIGURE 28 The WINIPCFG.EXE display

If your IP address is given as 0.0.0.0 then something is wrong with the IP configuration on your server. Sometimes you can make it work by pressing the "Renew All" button.

Windows® NT™

Use the Start menu on the server, select the Programs sub-menu and then the MS-DOS prompt command to create an MS-DOS window. Type the command

```
ipconfig -all
```

And press return. No new window will appear but you should be able to read the IP number from the MS-DOS console.

15. Appendix 4 Examples of expression syntax

Correct syntax

Expression	Expected result	Result	Notes
1	TRUE	TRUE	
0	FALSE	FALSE	
!0	TRUE	TRUE	
1=0	FALSE	FALSE	
1==0	FALSE	FALSE	
1!=0	TRUE	TRUE	
30	TRUE	TRUE	
3<0	FALSE	FALSE	
3&&0	FALSE	FALSE	Any integer that is not 0 is treated as TRUE
3||0	TRUE	TRUE	
1||0||1	TRUE	TRUE	
1&&0&&1	FALSE	FALSE	
(1&&(0||1)||1)||0	TRUE	TRUE	
(1&&0)&&(1||0)	FALSE	FALSE	
"twenty"="twenty"	TRUE	TRUE	
"Twelve"!="twelve"	TRUE	TRUE	String comparisons are case sensitive
"a"=="b"||121	TRUE	TRUE	
("a"=="b")||(121)	TRUE	TRUE	
"apple"=="banana"&& ("fig"=='fig')==1	FALSE	FALSE	

Incorrect syntax

Each expression in this table has one or more syntax errors. The results of these expressions are undefined and (ideally) there will be an error message for each one. Note that, if any of these errors affects a student, no visible message will be generated AND the expressions will be treated as evaluating to FALSE. It is up to staff to properly test all documents and eliminate this sort of thing.

Expression	Result	Explanation
word	(*** Missing operator in expression "word" ***)	Does not evaluate to 0 or 1
1+3==4	(*** Missing operator in expression "1+0" ***)	Arithmetic operators are not supported at the moment. This feature may be added to a future version.

"value"==4	(*** Missing string comparison operator in expression ""value"==4" ***)	Strings can't be compared directly to numbers
"apple"<"banana"	(*** Bad operator found in string expression ""apple"<"banana"" ***)	Sort order operators not supported (yet).

16. Appendix 5 Exercises to help you get started with the Fish

This document is a supplement to the FlyingFish user manual and software release on CD-ROM. It is intended to help you with a range of administrative and authoring tasks you might reasonably want to do.

You may find it helpful to work through the exercises in the order given: it will help you become familiar with both the software and the user manual.

To use the CD-ROM you will have to be able to unzip some files. if your computer does not already have a zip utility then you should start by installing PKZip. An installer is provided on the CD-ROM.

16.1 Basic server installation

1 If you already have a c:\FlyingFish directory, rename it to c:\~FlyingFish so that it will not interfere with later steps.

2 Unzip the FFish139 package from the CD-ROM (it's in the Win directory).

3 Install course material for Dynamics_100 into the directory c:\FlyingFish\task\Dynamics_100

4 Start the FFish139 server and log in as described on page 6 of the user manual.

5 Create a user identity for yourself (page 25, section 5.3)

6 Log in using your new identity.

7 Load the Dummy class from the text file (DummyCourse101.txt) provided on the CD (page 26, section 5.4)

16.2 Editing web pages and creating new ones

8 Modify the front page using the on-line editor, so that it contains a link to a new course called Dummy101. Use the link marked "Edit this file" at the bottom of the page – NOT the editor built into the Browser (page 14, section 3.8).

9 Use your new link to "go" to the non-existant course and create a page at the new location.

10 Use the directory and edit functions to create and modify the file c:\FlyingFish\task\Dummy101\info. Add a key like this:

Student=@?(u){/user/Course/DummyCourse101}

11 Restart the FlyingFish server so that it shows your new course in the monitoring window.

12 Use the monitoring window to navigate down to Dummy101 in the monitor and check that only the Dummy students appear at that level.

13 Create a new problem set at the Branch

Dummy101/set1

with sub-directories for three practice questions and one assessed question.

14 Set up the set1/body.html file so that it shows the full name of the student in large letters at the top.

15 Set up the info/Complete records for Dummy101 and set1 (page 40, section 7.1)

16 Create a problem in each of the directories you just made e.g.

Dummy101/set1/PQ_1

You can recycle any problems available anywhere in the existing sets. If you copy a problem directory to your new set, the job is mostly done...

Examples of interesting problem types can be found in Authoring/ and Dynamics_100/

17 Provide links to the practice and assessed questions in the set1/body.html document. (page 20).

18 Check that you can solve the problems and use the "Go to Next" function.

19 Set up access restrictions so that a student must solve the practice questions BEFORE the assessed question in set1. (page 41).

20 Add a "Congratulations" message to your set1/body.html document, which only appears if the student has finished the assessed question. (page 32, section 6.4)

16.3 Create a new "drag-and-drop" Java problem

Sorry – this section is not yet ready and I will give out supplementary notes.

16.4 Create a new "single numerical answer" Java problem

Sorry – this section is not yet ready and I will give out supplementary notes.

16.5 Create a completely new type of Java problem

Sorry – this section is not yet ready and I will give out supplementary notes.

17 Index