Writing Interactive Fiction In PAWS

This document is intended to be read *in addition* to the *PAWS Technical Manual* that comes with PAWS. Definitely read this document first, but don't throw away the documentation that comes with PAWS! Although written with a technical bent it has information you may need.

INTRODUCTION

Who Should Read This Book

This is the book you should start with, it teaches you how to write a game in PAWS and doesn't get too technical. If you want a technical (programmer's) overview of PAWS you should examine the *PAWS Technical Manual*, but we suggest doing so only after reading this book. This book gives a good grounding in PAWS that will help you navigate the technical manual—which you'll only need to do if you're a programmer who wants to extend PAWS itself rather that just writing games.

A Brief History

A long time ago and a long way away a new type of game was invented for the computer. It was called Advent and it was unlike anything ever written for a computer before. It's been endlessly copied down through the decades since its creation. You can find a version of it that will run on nearly every computer ever built, from the smallest microcomputer of the 1980's to the latest Cray super computers.

Advent (also called Adventure, or the Colossal Cave) was written over 20 years ago in a programming language called FORTRAN. By today's standards it was crude. You could only type in one or two word commands, and it would only recognize the first 6 letters of each word as unique.

And yet, and yet... Advent took university campuses by storm. Tens of thousands of students and professors burned up millions of computer hours delving deep into a dwarf-infested cave, with only their trusty brass lantern between them and certain doom...

I was fortunate enough in 1978 to obtain a copy of the original (350 point) version of Advent. The program listing was over two inches thick, in spite of the fact most of the program was actually stored in data files. Being a programmer (and egotistical!) I thought I could do it better. And so Quest was born in April of 1979.

Quest, alas, never shared Advent's popularity. Quest was only available for DEC PDP-11's, and was never really finished.

About that time the prototype of what was to become Infocom's Zork came out. It was called Dungeo and simply blew away Advent when it came to intelligence. You could actually type in real sentences of great complexity and Dungeo had as much or more of a plot than Advent did.

Despite all that, Advent (in its many later iterations) is *still* one of the most popular text adventure games of all time. In fact, Advent is the reason why interactive fiction was originally called text *adventure* in the first place!

In the 20+ years since Advent was written the tools available to writers of interactive fiction/text adventure have exploded. Advent's original language, FORTRAN, didn't even have text handling ability. It almost didn't have alphabetic capability at all! Much of that two inch stack of program listing was devoted to breaking down what the user typed in and making sense of it.

Today, interactive fiction (IF) writers are presented with a bewildering array of authoring languages, pre-written base libraries, and tons of example programs in their language of choice. There's even an Internet newsgroup called rec.arts.int-fiction devoted to writing them.

Be glad these resources are available to you. In the bad old days you would have been lucky if you could work on a dumb terminal that had a CRT instead of a Teletype machine. Always assuming you could beg, borrow, or steal time on a computer...

Why Choose PAWS?

There are a number of authoring languages you could have chosen. The more popular ones are TADS, Inform, Hugo and Alan. So why choose PAWS?

Let's be honest. PAWS does indeed have its flaws. It's a set of libraries on top of a real programming language (Python), which means you'll be writing some actual (gasp!) programming code. PAWS tries to eliminate 99% of the work, but you still will be programming—even if you don't recognize it as such.

PAWS has several advantages in my opinion over traditional IF languages. First, the syntax (the rules on putting language statements together) is much less rigid than other languages, and there isn't nearly as much *syntax sugar*. Syntax sugar consists of special characters used to make the *compiler's* job easier. For example, in TADS each line of code must end in a semi-colon (;) and blocks of text must be included in curly braces ("{" and "}").

For humans, syntax sugar isn't nearly so sweet, especially for inexperienced game authors. Forget a single semi-colon and you could spend hours trying to figure out why a TADS game won't compile.

In PAWS (Python) blocks of code are simply indented, like a story outline. And lines are ended as you'd expect—when the line ends! There are a couple of natural exceptions, like triple quoted text, however those few are easily learned and intuitive.

Chapter 1
Getting Started

You wouldn't be reading this if you didn't want to write some interactive fiction. You have an idea for a new game, right? Be it a western, science-fiction, fantasy, police-action, or whatever. You have an idea...

Turn Off The Computer!!!!!

Surprised? You shouldn't be. Think about it. What is the computer going to do for you in terms of shaping your story? Will it let you envision that haunted graveyard? Will it give you inspiration for the tragic girl who's sitting across the desk from the player, sobbing over her dead parents?

Will it let you paint a word picture of the majestic mountains, fading blue into the distance, topped with brilliant flashes of ghostly white?

At best the computer will let you enter your story on a word processor. This is *not* PAWS programming we're talking about, it's just plain old-fashioned *writing*.

Credit Where Credit Is Due

Most of the ideas and suggestions presented below aren't mine. They've been taken from the Alan manual's chapter *Adventure Construction*, from Graham Nelson's excellent *The Craft Of Adventure* articles, and (a few) from my own experiences with writing adventure games.

As an aside, *please* read *The Craft Of Adventure* before setting out on your IF odyssey, it will give you a huge leg up when creating your game! You should be able to find it at http://www.weblint.org/~neilb/intfiction/craft/ or at the IF Archive ftp://ftp.gmd.de/if-archive/ or at the Los Angelos mirror of the IF Archive ftp://ftp.ifarchive.com/if-archive/. The IF Archives are the Rome of the IF world, all roads lead there. You'll find an enormous amount of material there, including other IF languages, how-to articles, and other goodies.

While we're on the subject, Roger Firth (no relation ☺) maintains an excellent site of IF resources at http://homepages.tesco.net/~roger.firth/, including a page that implements a small game called *Cloak Of Darkness* in several different IF languages, it's a good way to compare languages and see if PAWS really is likely to be what you want to use.

The Starting Point

Let's begin with a quote from Graham Nelson:

> *'In the beginning of any game is its 'world', physical and imaginary, geography and myth. The vital test takes place in the player's head: is the picture of a continuous sweep of landscape, or of a railway-map on which a counter moves from node to another? 'Adventure' passes this test, however primitive some may call it. If it had not done so, the genre might never have started.'*

This is (in my opinion) the real starting point of a game. Create your world, but do it *inside your head*.

The single most important criteria of any game is that it be, in some sense, *real*. Players have to be able to suspend their disbelief while playing your game. There are many ways to achieve this feeling of reality, but a few are particularly important.

First, you have only words on a screen to work with. Unlike the graphical adventure *Zork Nemesis*, for instance, which is a modern multimedia masterpiece involving surround sound, 360 degree pictures, live actors, animation, and other multi-million dollar technologies, all you have to work with is text.

That—and the human mind, without doubt the greatest virtual reality machine in existence. When creating your own 'virtual reality' what you have to do is fire the player's imagination.

The first tool is one that storytellers have been using for thousands of years—evocative wording. Obviously a skilled writer has the advantage here. A few well chosen words can evoke fear, haunting beauty, laughter, or tears. There's no real substitute for good writing, it's the foundation on which your adventure will be based.

Making a Masterpiece

What if you aren't a skilled writer? In that case you might want to pick up a few books on writing. They can offer basic writing skills and point out areas where you could improve your writing. It isn't a panacea but it will give you a place to start.

In addition, there's another way to improve your writing, one that's very effective. Actually *create* your world inside your head. See it, smell it, taste it! Take time to answer many basic questions about it. The most trivial detail often adds verisimilitude to your writing. What does that maid like to have for breakfast? Is the blacksmith color blind? Is it spring in your world?

No matter what style of game you're creating you've got some really basic questions to answer.

Setting

First, the setting of your game. Is it an abandoned cave like in Adventure? If so why was it abandoned? Who made it? Why are there treasures scattered hither and yon in it?

You're going to do a lot of background work that never shows up directly in the game. For example, Adventure is based on an area of Mammoth Cave in Kentucky. Willie Crowther spent a lot of time exploring Mammoth cave. When it came time to write Adventure, he already knew a lot about the setting since he'd spent a long time there.

It came through in his writing. When he described the canyon for example he was describing a place he'd already *been*, in fact he probably went back there and took notes!

Obviously if your setting is fictitious you won't be able to describe a real place. But if you've spent a long time thinking about a place, to you it is in some sense real.

My wife (who has a somewhat mystical turn) often says '*The more you dwell on something the more energy you give it. Give it enough and it will become real, somewhere.*'

Plot

If a place is the *setting*, then what happens there is the *plot*. There are two driving forces behind plotting a story. First, what goal is the player trying to accomplish? Is finding out the goal part of the goal itself? In Adventure the plot was simple, get all the goodies and go home. In Myst, another classic (graphical) adventure the main goal is again to get back to where you came from—but you discover, as you play, there's a much darker and more dangerous plot that you've been caught up in…

Second, what stands in the way of the player accomplishing their goal? In Adventure there were all sorts of geographical barriers (locked doors, cave-ins, chasms), creatures (homicidal dwarves, greedy trolls, hungry bears), and puzzles (how to catch the bird).

Actors

Unless your world is sparsely populated (like Adventure) you're going to have actors in it. What are *their* goals? In a murder mystery game, for instance, who actually killed the victim and what was their motive? What are they likely to do when the player starts snooping around? Was the murder done at the bidding of some big shot? If so, why?

Actors *make* the plot. It's the conflict between the player's goal and the actors' goals that make up the story. After all, if the game is about saving the world, but the world's in no danger you don't have much to do…

Summary

In short, you have to do a lot of legwork crafting your story's background. Why is the setting the way it is? What's the player doing there in the first place and how do they extricate themselves from the mess about to befall them? Who's out to stop them from doing so? Why are they trying to stop the player?

A good rule of thumb is that the player will never know more than 5% of what you know about your world. Did you know, for instance, that Infocom created a full color map of the Great Underground Empire and offered it for sale to players? I saw a copy, it was a gorgeous thing. Showed all sorts of interesting proximities that made perfect sense when you saw a map but that you would never have guessed from playing the game.

Take it for granted that knowing what the girl's brother's taste in cars is won't make it into the game. All the seemingly worthless background you create that never makes it into the game will still color every part of the game and give it a sense of reality it wouldn't otherwise have. It still won't compensate for terrible writing, mind you, but it *will* lend your writing a much needed boost, be you ever so skillful.

Finally, read chapter 7 in the Alan (not PAWS!) manual, the first two sections (*Getting An Idea* and *Elaborating The Story*) offer excellent suggestions on how you can go about this. You can find the Alan manual at the IF archive.

From Story To Program—Sort Of

The next step is to figure out how your story is flawed from an implementation standpoint. This is a *lot* trickier than it sounds, but surprisingly it doesn't require you to know anything about PAWS yet!

Again, from Graham Nelson:

'It must never be forgotten that they (adventure games) intentionally annoy the player most of the time. There's a fine line between a challenge and a nuisance: the designer has to think, first and foremost, like a player (not an author, and certainly not a programmer).'

In other words, it's time to check your puzzles to make sure you haven't broken what Graham Nelson calls the player's bill of rights:

1. Not to be killed without warning.
2. Not to be given horribly unclear hints.
3. To be able to win without experience of past lives.
4. To be able to win without knowledge of future events.
5. Not to have the game closed off without warning.
6. Not to need to do unlikely things.
7. Not to need to do boring things for the sake of it.
8. Not to have to type exactly the right verb.
9. To be allowed reasonable synonyms.
10. To have a decent parser.
11. To have reasonable freedom of action.
12. Not to depend much on luck.
13. To be able to understand a problem once it is solved.
14. Not to be given too many red herrings.
15. To have a good reason why something is impossible.
16. Not to need to be American.
17. To know how the game is getting on.

Most of the rules boil down to common sense, or courtesy. For instance it isn't fair to kill off a player without warning, and it often spoils their enjoyment of the game—which means they won't play it again!

You have to analyze your puzzles. Are they too hard? Are the clues too cryptic? On the other hand are the clues a dead-giveaway? Only rewriting will correct these kinds of flaws (and trust me, you'll be rewriting a *lot*).

Another important point of the bill of rights is that games should follow an internally consistent logic. No matter how bizarre it may be, as long as there *is* logic players will be able to predict the consequences of their actions—an important factor in keeping their interest. Don't hit them in the face with 'the laws of the universe', make sure they have to work to figure out the game logic, but make sure they can. The sense of accomplishment this gives players will ensure they remember your game fondly when they talk about their favorites.

One of the subtlest flaws a game can have is one I call 'you can't get there from here'. Basically a flaw like this means that unless you do something right at the beginning of the game you get stuck half-way through with no chance of correcting the mistake you made at the beginning. ***THIS ANNOYS PLAYERS ENORMOUSLY!*** The graphical version of 'Return To Zork' suffered from this malady.

One important point to any text adventure is the *parser*. The parser is the engine that decodes what the player types into nouns, verbs, etc. Fortunately PAWS has a quite sophisticated parser so this isn't much of a problem.

What *can* be a problem (and one we'll cover later) is a limited vocabulary. Players are notorious for typing the most unlikely commands. Instead of a nice simple 'Take ring' they're likely to type in 'pick up ring' or 'get ring' or even 'steal ring'. As the game author it's up to you to supply all the vocabulary for the game. This is one area you'll spend a huge amount of time in.

Another problem area authors fall into is the 'linear plot' approach. This basically means the player has to play following a prescribed sequence of steps in exactly the right order or they won't progress.

You would be better advised to create a 'plot tree'. This means the player can try to solve one problem and if stymied can turn to another. This keeps them from getting frustrated and quitting the game altogether. Remember, your task is to annoy the player to the limit of enjoyment—but no further.

And finally remember that people all over the world will be playing your game. It's very difficult to anticipate when cultural differences will cause misunderstanding or bad feeling, but it can happen. Or, more likely, what a person in England would find hysterically funny a person from Russia (with English as a second language) wouldn't find funny at all. This is particularly true with puns, which don't translate well.

Here's one example I can think of off hand. In Russian the colloquial name for a collapsar is 'frozen star'. Americans call such a stellar body a 'black hole', which in Russian is a very vulgar term. Another example, the word 'nova' in English is very evocative (an exploding star), but in Spanish the word translates to 'doesn't go'.

And, of course, always remember than British and American versions of English are *different*. This is particularly true for slang terms and colloquialisms.

Again, for a solid grounding in plot line and story development read Graham Nelson's *The Craft Of Adventure*, it's a superb primer for plotting adventure games.

Chapter 2
Setting Up PAWS

Since you are reading this I'm going to assume you know how to do basic housekeeping on your system. I personally use Windows 2000 on a PC, you may use a Macintosh, Amiga, or some flavor of Unix system. So I won't try to give a detailed plan of set up, I'll just tell you what needs to be done and trust you know how to do it.

Getting The Compiler

First, if you downloaded PAWS from the Internet, you didn't get the Python language/interpreter (needed to translate your game into a program the computer can run).

Python is available for almost every computer and operating system ever made. To get a (free) copy for your particular machine go to the www.python.org website. They have a download section, simply go there and download Python for your particular machine and OS. Although PAWS was designed with Python version 1.5.2 it will also run in versions 1.6 and 2.0 of Python as well.

For mainstream operating systems (Windows, Mac, and Linux) you'll probably have no difficulty finding version 1.6, or even 2.0. However, if your computer runs OS/2, a less popular version of Unix, or other more exotic OS you'll probably have to get version 1.5.2.

Version 1.5.2 was the version I used to create PAWS originally, PAWS/Universe design was kept deliberately simple, so earlier versions than 1.5.2 might actually work, but I haven't tested them.

The documentation that comes with the Python download will tell you how to install it. On a Windows machine, that basically involves double-clicking the download executable. On a Macintosh the SIT file should automatically expand to provide the Installer.

Other operating systems will use different installation routines, see the Python documentation for your individual OS.

Folder structure

Remember, PAWS is available for a lot of different operating systems, over 20 at last count! Therefore I'm going to use Windows as my example system. Macintosh users should have little trouble following the discussion below. UNIX/Linux users should substitute the word "directory" for "folder"

In Windows versions of Python at least all you have to do is create one folder to hold your games. I call mine *APPS*, but it really doesn't matter what you call it or where you put it.

In the *APPS* folder create one folder for each game you plan to write. Then copy the following files into *each* folder: *PAWS.py*, *Universe.py*, and *play*.

Installing The Thief's Quest Sample Game

To install Thief's Quest, create a folder for it and copy *PAWS.py*, *Universe.py*, *play*, and *TQ.py*. *TQ.py* is the actual game, you'll notice it's the largest of the files.

In addition for Windows or DOS systems also copy the file *Thief's Quest.BAT*. If you're using an earlier version of DOS that doesn't support long file names you can rename the batch file *TQ.BAT*.

Finally, if you plan to use the ScopeEdit text editor (Windows only, sorry), copy *Shell.se* as well. You should also copy the *Python.SEO* file to the ScopeEdit directory.

Tools Of The Trade

A writer of traditional fiction needs a word processor, right? As a writer of interactive fiction you need something different, a *programming text editor*.

Programmer's editors are specially designed for writing programs—and make no mistake, despite the fact PAWS is advertised as an easy IF tool, you're still going to be programming. That might scare you a little, after all you probably chose PAWS so you wouldn't *have* to program. You just want to write interactive fiction, right?

Well, programming isn't that bad (especially in PAWS). And a good programmer's editor can make all the code writing easier. My personal favorite is called *ScopeEdit* (unfortunately *ScopeEdit* is only available for Windows). *ScopeEdit* is a shareware program that costs $79 U.S. but it's worth every penny! You can find *ScopeEdit* at www.loginov.com on the web. They have a demo version so you can try it before actually putting out any cash.

ScopeEdit has several advantages. First, it does *syntax highlighting*. This makes text change color appropriately. For example, my setup has comments appearing in gray (so they fade into the background) , keywords in blue, functions in red, and quoted text in orange.

This might seem trivial, but it makes reading a PAWS program *much* simpler. In addition, it allows you to catch spelling and capitalization mistakes immediately, since the word won't change color. That alone is worth its weight in gold because PAWS is case sensitive: in addition to spelling correctly you also have to capitalize correctly. If the word you type stays black when you expected it to change color, you know you mistyped it.

ScopeEdit also supports *file branching*. This means editing multiple files from within one file. This is a godsend in PAWS because you can have all the pieces of your game in one place and simply click to go from one piece to another.

Another feature of *ScopeEdit*, called *code nesting* or *folding* allows you to hide portions of your program, viewing only one small part at a time. This makes it easier to see the forest and avoid looking at the individual trees until you want to.

I've included an *SEQ* File for PAWS along with this document. The *SEQ* file can be copied to the Program Files\ScopeEdit directory then associated with the .py file extension to properly colorize your text. You may obtain a shareware copy of *ScopeEdit* from the Internet. Install it on your computer as you would any other Windows program.

For the game author on a budget you may want to examine *Origami*. This cheaper text editor runs about $20 for the registered version, it supports limited syntax highlighting (255 keywords only) and folding.

Text Editors For Other Operating Systems

The Macintosh has an editor called BBEdit which supports syntax highlighting but not folding. However since syntax highlighting gives you the most bang for the buck this isn't a huge issue. You'll have to create your own syntax highlight file for BBEdit, however, since I don't use the program.

I'm not very familiar with programming editors available for other operating systems although I do know Origami is available for Unix/Linux systems. Other editors exist, ask around among the gurus of your particular OS. Be aware, however, that text editors (like any other favorite piece of software) are often the subjects of heated debate! Everyone has their favorite. As minimum features, however, look for the ability to color text (customizable to different programming languages!), and the ability to open multiple files at one time. Text folding (common in Linux/Unix editors) is also a *big* plus.

Chapter 3
Thief's Quest

You knew it was coming, didn't you? Every tutorial ever written about programming comes with an example program, usually the author's pet project.

Well, this tutorial's no different. Included with this documentation is the complete source code to *Thief's Quest*, a huge game that's been extensively commented (and documented here) that you can use to explore all sorts of PAWS program tricks, from the simplest to the most complex. Note that Thief's Quest is nowhere near finished, I decided to release PAWS before the (enormous) TQ game was completed so others could use it.

True Confessions

I'll be honest with you. I'm writing this book at the same time I'm writing Thief's Quest! That way I'll be able to tell you about all the problems I stumbled across so you won't have to. Also, *Thief's Quest* has never actually been finished in any of its incarnations.

The closest it ever came to being finished was in the original version, written in 1979. that version had an incomplete level 6—no version since then has gone beyond level 4! So in effect I'm going to be writing levels 5 thru 7 from scratch, as well as the end-game.

And you'll come along for the ride, we're going to explore TQ as I write it, showing you everything. A game 20 years in the making will finally see the light of day!

It almost feels like an archeological dig, doesn't it?

History In The Making

In 1978 I was a high school senior who was able to take advantage of a local university's computer-access 'guest' program. They made their computer lab available to high school students when it wasn't otherwise required for their own students.

I was already head over heels in love with computers, and swore one day I'd have my very own. (In 1978 that was nothing more than a wild fantasy, the kind of computer I wanted (like the one at the university) cost at least $100,000).

It was here I discovered Advent, the very first text adventure game. And like countless others before me I was hooked. I spent endless hours glued to that terminal, ferreting out the secrets of the cave, dodging dwarves, squirming through virtual tunnels, slyly pocketing the odd silver bars…

I was at the same time learning how to program in FORTRAN on an antiquated (even by 1978 standards) high school computer. (Remember the old punch cards? This thing read them. One at a time. *Slowly*.)

By the time I graduated I was firmly convinced that programming was going to be my avocation as well as my profession.

As it happened I was able to keep using the university computer, even though I wasn't a student (university or high school). The summer of 1978 also saw my high school throw out the old computer and get a shiny new PDP-11/45 complete with 7 terminals…)

It's hard to understand just how difficult gaining access to a computer was in 1978. Nowadays you can get a used computer for as little as little as $20 if you don't care about sound cards and CD-ROMS and Pentium performance.

But in 1978 there were *no* personal computers. The PDP 11/45 was a mini-computer, a small mainframe that still leased for $2,000 dollars a *month*. In other words it represented a huge investment. The school used it to run their payroll, academic requirements, and teach students computing.

But if you were patient, and willing to come in at odd hours and throw in free programming services you could get access for your own projects.

One of which was playing *Dungeo*, a freely distributed game on PDP-11 computers at the time. *Dungeo* was like Advent, only *better*. It understood complex commands and had an axe-wielding troll and a Cyclops and all kinds of nifty new gizmos to play with and places to explore…

The First Incarnation

Show a programmer a world-shaking program like *Dungeo* or *Advent* and I'll guarantee you they'll think of a thousand ways *they* could do it better.

And so, *Quest* was born. In those days it was just plain *Quest*. The parser was only so-so, it was an improvement on *Advent*, but that isn't saying much. It could understand words of arbitrary length (Advent was limited to 6 significant letters), it could discard articles like *a*, *the*, or *an*, it could handle multiple commands separated by *and*, and it understood *all*. But it was still just *verb/object* and *Dungeo's* parser could eat *Quest* for breakfast.

Quest was at first a shameless copy of *Advent*. The first 5 rooms in the cave were almost a total replica of *Adventure*. But by the time I expanded the first level I was breaking away from *Adventure*, incorporating my own ideas and letting my fevered imagination run wild.

By the time level 4 rolled around I was collaborating with another programmer and he did almost all of level 5's design. (A very *Hobbit* kind of feel, complete with Gollum and a riddle contest.) Gollum was in fact an actor, probably one of the first in the genre. He didn't do much except appear, ask riddles and disappear (or eat you if you missed a riddle) but he was actually an actor in today's sense of the word.

Level 6 was never completed, it was a very scenic medieval kind of place, full of traps, and had a dragon. But the real world intruded, and my interest waned. Understand *Quest* was written in Basic-Plus, which meant every level was a program. It was *hard* to write, and often tedious.

In other words I plain ran out of gas.

The Second Version

The world doesn't stand still. Computer technology advanced at warp speed and soon CP/M appeared, running on computers that cost about $2,000. They actually had *floppy drives*. My fantasy had come true. I bought my very first computer, a Kaypro II with 2 floppy drives (holding a whopping 191k each) and 64k of memory!

I was in heaven. Understand that in 1982 this was state of the art stuff, at least for the common man. The Kaypro II was considered a 'business' computer, intended for serious applications, running accounting programs and so on.

Naturally I thought about *other* kinds of programs. Like *Quest*…

The second version of *Quest* was written in S-Basic (Structured Basic), a language I still regard as one of the finest dialects of Basic ever written. By August 1983 I had mastered S-Basic and could make it turn handsprings. Even so, with every memory saving trick I could squeeze out of the language (and there were a *lot*) *Quest* still strained the machine. So *Quest* stopped at level 4, slamming face first into the hard wall of too much program and too little computer.

But I'd proven I could do it and that was enough.

Land of the Infinite Parentheses

The third version of *Quest* was written on an IBM-PC AT in November of 1988. Five years after CP/M there were no longer technological hurdles to implementing *Quest*, the AT had 10 times the RAM and there was even a language specifically designed for writing adventure games!

It was called ADVSYS. Version 1.2 had just appeared, the last (and still most popular) version of the language.

ADVSYS had no library and pitiful documentation. It was based on LISP, a truly bizarre language that has the dubious distinction of being easy to write but impossible to read. So onward I slogged, creating the library of routines for such things as moving, taking objects, handling light and dark rooms, sound, and smell.

The only thing that saved my sanity was that ADVSYS is object-oriented, meaning it did get easier once the groundwork was laid.

However, time and interest soon disappeared and this version halted with an incomplete level 3.

However, the ADVSYS version had a parser than understood many constructs. It could handle adjectives, the first version able to do so. It could handle multiple object gets and drops, in short the parser was as good as the one found in *Dungeo* (later to be made famous as *Zork*).

Alan

The next version was an exploration of Alan, which ended before it began. Alan wasn't bad, but it wasn't a programming language and I found its constraints too confining. This was perhaps the most limited version of TQ, with only a couple of rooms.

Hugo

The next version was started in Hugo in September of 1996, and renamed *Thief's Quest*. Hugo was a vast improvement on ADVSYS, with a better syntax than either TADS or Inform (in my opinion). However, it proved to have inadequate documentation for the library (a fatal weakness in any software program these days) and proved to be more work than I had time for. The Hugo version has an incomplete prologue, with just a map going from place to place.

TADS

One of the most popular of today's IF languages, I actually crafted most of the surface level of TQ using the WorldClass library. WorldClass had some disadvantages to my mind, so I created a new library called Universe that worked more like the way I wanted it to. I also created a visual editor/librarian program called RAVEL, and again got most of the surface finished.

I liked TADS, but all the compiler's syntax sugar drove me nuts. TADS is based on C++ style syntax, which I've never really cared for.

PAWS

Which brings us up to the present. Not finding any of the existing purpose-built languages suitable I turned to a general purpose language called Python. I then created the parser and "world" libraries required to actually write a game in Python. Collectively, the parts I created are called "PAWS", and they will allow you to easily implement the game you've finished writing in your head. (You *have* finished it, right? ☺)

The Plot

Thief's Quest has a simple plot. You're a thief who broke into a wizard's tower to see what kind of magical goodies you could lay your hands on. But you made a *serious* mistake. The wizard happened to be home at the time and caught you!

The wizard declared the punishment should fit the crime, so you end up in a rift valley completely surrounded by cliffs that can't be climbed. Your goal is simple.

Get Out.

Of course if you can pocket the odd treasure or two on the way to the egress, all the better. But then again, crime doesn't pay, does it? (cue sinister laughter…)

Chapter 4
Actually Implementing A New Game

Thief's Quest may never have been finished, but it does follow the golden rule. Work out the game in detail before writing one line of PAWS code! Each incarnation of the program gained new ideas and new insights, with new abilities gained. After 20 years stewing in my subconscious it's as ready as it will ever be.

Whether it's as good as I think it is, well, that's another matter entirely!

How Long Will It Take?

Please understand I'm not telling you it will take you anything like 20 years to create your own game! Usually a first game (of normal size) will take about 6 months, from idea to beta-testing.

Beta-testing is where you give out the game to a few willing people and let them tear it to shreds. Then you pick up the pieces, go off for a good sulk, and put the pieces back together, fixing the problems.

And then you give it back to the beta testers… Two or three more times.

If you have more time to devote to your game obviously it will get done quicker. The six month figure is for folks working two or three hours one or two nights a week, with perhaps another 3 hours on a quiet Sunday afternoon.

Some Housekeeping

Let's pretend for a moment that we are starting from scratch. We're going to assume you have found a text editor you like and you know how to use it. We also assume that you have installed Python.

1. Did you create a games folder? If not, do that now. I called mine *APPS* but you can call it whatever you like.
2. Inside your newly created games folder, create a new folder. In Windows 95/98/NT you can use a long file name, so I called mine *Thief's Quest*. The Macintosh and Unix/Linux systems have different rules about folder (directory) names, so you may not be able to use the apostrophe.
3. Copy *PAWS.py*, *Universe.py*, and *play* into the *Thief's Quest* folder.
4. Using your text editor, create an empty text file called *<game name>.py*. In the case of Thief's Quest this would be *TQ.py*. In our Colossal Cave example it would be *CC.py*. Remember that case is important! In other words, *TQ.py, Tq.py, and tq.py* are treated as 3 separate files by Python.

Creating A Script/Batch File

This step is for DOS, Windows, and Unix/Linux users. Macintosh users won't use Python the same way, see MacPython's documentation for the equivalent step. We need to create a batch file (script file in Unix) to run Python.

I'm going to create a file to run Python and make sure it's looking in the right folder. If you're using Linux/Unix you'll have to substitute the appropriate shell script commands, because frankly I'm not familiar enough with shell scripts to even guess. ☺

Here's the batch file I use. I call it *Thief's Quest.BAT*.

```
1. CD "C:\APPS\Thief's Quest"
2. "C:\Program Files\Python\PYTHON.EXE" play TQ
```

The line numbers are *not* part of the batch file, they're just there so you can relate the explanations below to the appropriate line. Also, please note that even though we're in Windows we're still using a *DOS* version of Python. Since the IF that PAWS is designed to create is text based, we don't need the added complexities of Windows Python. Mac users will see the equivalent of a text based screen, even though technically MacPython is graphical.

1. Line 1 changes to the current folder. Python looks for referenced files in the current folder first, and then along a predetermined path set during Python's installation. This command makes sure it knows where your game files are.

2. This runs Python, which automatically runs *play*, which in turn will run *TQ*. It may seem a bit involved, but remember Python is a general purpose language. In fact, once you learn Python you can use it to write any kind of program!

And Now The Main Event!

Ok, you've been patient and followed all the directions. Now, dash it all, you want to write your game!

I'm going to show you how, by referring to the *Thief's Quest* sample game. Remember, you need *PAWS.py* and *Universe.py* and *play* for any game you write. These are the "laws of physics" for your game's world.

Your actual game is *TQ.py* (or *CC.py*, or *Zork.py*) or whatever you happen to name it. This file contains all the text and programming for your game. We recommend keeping the file names short since your game might be played on an operating system that doesn't support long names or names with spaces in them.

Chapter 5
Start At The Beginning

We're going to take this logically, and show you the pieces you need to create in the order you need to create them.

Look at *TQ.py*. If you use a regular text editor you'll see a massively long listing. That's ok, just read it starting at the top we won't go very far down in this chapter.

If you have ScopeEdit, on the other hand, you'll see a single page that starts with some gray text and ends with some gray text.

Note: When you write a PAWS program you're actually writing a *Python* program. Python is the programming language PAWS was written in. PAWS is actually just a set of libraries for Python, a sort of pre-written rule base for your game.

Therefore in the discussion below we tend to use "Python" instead of "PAWS" when talking about writing your game unless we really are talking about one of the files (*PAWS.py*, *Universe.py*, or *play*) that makes up PAWS.

Comments Are Your Friend

Take that to heart. Comments are a way to make notes to yourself in the game as you write it. *Thief's Quest* and PAWS in general is *very* heavily commented, in fact comments make up about ***70 percent*** of PAWS and *Thief's Quest*!

There's a reason programmers call what they write source *code* or just plain *code*. Like code, programs tend to be very cryptic. There are ways to make source code more readable, but the whole point of source code is to be unambiguous to the *computer*. Generally, readability for humans isn't even a distant also-ran.

That's why comments were born. Comments let you explain *why* a piece of code is written the way it is.

Six months after you write a game you'll come back and have absolutely no idea how it works. That's a guarantee. When you write code it doesn't make it into your long term memory you see. That's why you need to be very clear and thorough when commenting. In effect, you write *two* programs, one in Python for the computer, the other in English for humans.

Comments begin with a number sign symbol, "#". If you have a syntax highlighting editor you can make the comments any color you like.

If you have ScopeEdit and installed the *Python.SEO* file then you'll see comments in gray.

Header/Footer Comments

The first thing to do is create a header similar to the one in *Thief's Quest*. It's basically a box outlined with asterisks that gives the game's title, your name, copyright year and a (very) short description of the game. This comment box is a label for the entire game.

At the very bottom of the file you'll find another comment, this one saying "End Of Thief's Quest Game Library". This just marks the end of the file. It's helpful to mark the end of the file so readers know the file wasn't somehow chopped off in the middle.

Notes/Development History

I put a Notes section in my comments that follows the header. If you have ScopeEdit, just click the purple <Notes> to see the notes, then click the button that looks like an up arrow on the toolbar to return (the tool tip is "Up One Level").

Notes contains a more detailed plot for your game and any other general purpose notes you care to put there. Notes are optional, leaving them out won't affect how the game runs, but it will make it harder to find and fix problems later if you don't put it in.

The *Development History* section of comments allows you to create a history of changes to the program, along the lines of "Version X fixed these problems, discovered by John Smith. Released 1/1/2000."

Notes On Color Coding

You may hate the color scheme I've come up with, or your editor may support different background colors that you want to use to distinguish game specific items from PAWS library items. (Python language specific keywords and functions are already separate).

If so, then note the colors used by your editor in this section. You may also want to note which font and font size you used during development.

Chapter 6
Importing Libraries

If you're creating an actual game go ahead and save your work. It's always a good idea to "save early, save often". If you don't, chances are good that one day you'll lose hours of work, and the work you lose will be the work it's hardest to duplicate! Murphy loves the careless...

Including PAWS And Universe

At any rate you'll notice the next section is called *Import Game Engine And Universe Libraries*. The two lines of code you need to put in are:

```
from PAWS import *
from Universe import *
```

The first line allows you to refer to every object in the *PAWS.py* file as though it were actually defined in your game file. The second line does the same for everything in *Universe.py*.

As a side note, you'll notice that although we've reached line 180 or so of *TQ.py*, these two lines ***are the very first lines of programming code in the file!***

That's right! These two lines are actual program code, everything else up to this point has been nothing but comments! Your own games probably won't be as heavily commented, of course, but they probably should be.

Keep in mind that *PAWS.py*, *Universe.py*, *play*, and *TQ.py* were written with two aims in mind. First, *TQ.py* is aimed at teaching new PAWS game authors how to write games with PAWS.

Second, the other files were written with an eye toward teaching *experienced* programmers that use TADS, Inform, and Hugo how PAWS is put together, so they can create new libraries to extend PAWS, should they care to. It also helps them learn Python.

Summary

That wasn't too bad, was it? 2 lines of programming code that basically give you access to PAWS and the *Universe* library.

In the next chapter we'll cover your first *object*. If you're starting to get dizzy from all the new terms we're throwing at you, it might be helpful to take a break before we journey onward. Get a cup of coffee, put your feet up, and relax.

Chapter 7
The Game Object

Have a nice break? That's good, because we're about to lay a *really* heavy programming concept on you—the *object*.

What's An Object?

I'm glad you asked! An *object* is a thing. For example, a room is an object. A sword is an object. The player's character is an object. A *direction* is an object, for heaven's sake. Anything that can be thought of as a distinct thing is an object. A rock, for example, is an object. So is a monster. Or a tree.

Ok, now that you know what an object is, you should know that they have *properties*. Properties are things like the object's weight, its color, how big it is, and so on. That seems reasonable, right?

The object we're concerned about right now is the *Game* object. The game object holds information about the game itself. Here are the appropriate lines from *TQ.py*. (If you're following along in your own game, substitute the appropriate information).

```
Game.Author      = "Roger Plowman"
Game.Copyright   = "1978-2000"
Game.Name        = "THIEF'S QUEST - Getting OUT!"
Game.Version     = "8.0"
```

Scared you off yet? ☺

Seriously, there's nothing complex here. Notice how we use two words separated by a period? The period is called a *connector*. Basically, what we're saying in line one is that the *Author* property of the *Game* object should be set to the string "Roger Plowman".

The second line is saying the *Copyright* property of the *Game* object should be set to the string "1978-2000". And so forth.

You can guess the rest. *Game.Name* is the name of your game, as you want it to appear to the player. *Game.Version* is the version number of your game, normally you'll change this value to 1.0 instead of 8.0. I use 8.0 because this is the 8'th version of *Thief's Quest*. As you fix problems in your game you'll change this version number in .1 increments. So: 1.0, 1.1, 1.2, etc.

If you've ever used other computer languages, you can think of a property as a *variable*. It's a little different but basically used the same way.

New Concept: Triple Quoted Text

In Python there are 3 kinds of strings, two of which are interchangeable. For instance, you saw double-quoted text in the example above. You could also have used single quotes, if you liked, and gotten exactly the same result.

When should you use single quotes and when should you use double quotes? It really doesn't matter. If you have a string that contains apostrophes (single quotes) you need to use double quotes around it. For example:

```
String = "He's very strong."
```

If, on the other hand, you have a string that contains double quotes then you need to surround it with single quotes. For example:

```
String = 'The worm is 6" long.'
```

Most of the time you'll use double-quotes since most text has apostrophes. But there's a *third* kind of quoted text, one you're going to use a *lot* of—triple-quoted text.

Triple-quoted text allows you to include as many lines of text as you'd like between the triple quotes. It's very handy when you have a huge amount of text to put on the screen at one time—such as a room's long description.

If you look at *Game.IntroText* (ScopeEdit users click on <Game Introductory Text>) you'll see:

```
Game.IntroText = """
                 Example of triple-quoted text.
                 """
```

Well, actually you'll see a great deal of text, the example above is just for brevity. In the actual program everything between the two triple-quotes is *a single piece of text.* You'll also notice that a triple quote is made from 3 double-quotes in a row.

In other words, *Game.IntroText* is about 8 paragraphs long!

If you study the text you'll notice two things immediately. First, we seem to be breaking our indentation rule, the string doesn't line up with the triple quotes like in our example above.

This is OK, a triple quoted string ignores indentation rules. Second, each paragraph is separated by a blank line, as you'd expect, but each paragraph also ends with ~n ~n.

What You See Is *Not* What You Get

Triple quoted text has *two* appearances. The first is in your source code. You can format the text to appear neat and easy to read in your text editor's window. However, this is *not* how it will appear when the game is played.

When the game is played the *other* appearance takes over. This is governed not by how the text appears in your text editor, but rather how it appears to PAWS.

This double-standard allows you to automatically handle screens of different widths and (in *MacPython*) using windows of different sizes without worrying about ugly line wraps and other problems.

Smoke And Mirrors: The *Say()* function

This textual slight of hand is accomplished by the *Say()* function. In PAWS the *Say()* function replaces the normal Python *print* statement. *Say()* is much more intelligent than *print*, it does tricks *print* never heard of.

First, *Say()* handles the problem of keeping individual words from splitting on the right edge of the screen. If a word would extend beyond the width of the screen (and thus wrap to the next line) *Say()* automatically puts the word on the next line instead.

Second, *Say()* also handles the problem of having too much text to fit in the window at one time. If a single piece of text is longer than the screen, then *Say()* will pause, displaying a "[more]" message and not display the remainder of the text until the player presses any key to continue.

Third, *Say()* ignores line breaks and multiple spaces in text, treating all white space as a single space. For example, if your program said:

```
Game.IntroText = """
                 This is line 1.
                 This is line 2.
                 This is line 3.
                 """
```

The *Say()* function would print it on the screen as "This is line 1. This is line 2. This is line 3." (all on the same line).

Special Characters: Controlling *Say()*

Sometimes you don't want *Say()* to remove line breaks or spaces. This is where the *special characters* are used. You can see examples of this in the *Thief's Quest* intro text. The table below lists the special characters and what they do. Remember, all special characters must have a space in front of them and another space behind them. Thus "This is ok. ~n" is fine, but "this is wrong~n" won't work (no leading space before the ~n).

Character	Purpose
~m	This special character forces the [more] message to appear, whether the screen is full or not. This is very handy for long game introductions. If you don't force a [more] early, it's very possible to have the game banner scroll off the screen before the player can read it. The game banner lists the game's title and the copyright information. If you look at the *Thief's Quest* game introduction you'll see we use a ~m just after the words "fate is upon thee!". This is to keep the game banner from scrolling off the top of the screen. You'll have to experiment a bit with your own introduction text, especially if you have more text than will fit on one (or two!) screens.
~n	This special character forces a line break. For instance, if you have "line 1 ~n line 2" then the words "line 2" would appear on the next line, even if the current screen line weren't full. This allows you to force text to appear on the next line rather than on the same line. Note this is the equivalent of a carriage return/line feed combination, it puts text on the *beginning* of the next line.
~n ~n	Two line breaks in a row makes a paragraph break, a blank line between paragraphs. You'll notice that *Thief's Quest* uses this special character at the end of each paragraph in the intro text.
~t	This special character lets you print a *tab* character (actually 4 spaces instead of a real tab). This lets you indent lists (like the player's inventory).
{}	Anything inside curly braces will be treated as a Python variable. This is an advanced feature we'll cover later, it's quite easy to use (and very useful) but we won't cover variables for a bit yet. Trust me, we will explain it because it makes writing complex room descriptions *much* simpler.

What We've Covered In This Chapter

In this chapter you learned about the *Game* object and how to set its properties for your own game. You also learned about triple-quoted text and the special characters you can use in it. We also talked about the *Say()* function, although you'll become much more familiar with it in the chapters to come.

Feel free to take a break and review what we've covered so far. In the next chapter you're going to write a Python function, your first real Python code!

Chapter 8
Your First Function

You remember we mentioned that PAWS is a collection of pre-written programming libraries on top of Python? The primary reason to use PAWS is to avoid having to program when writing your game. However, sometimes (like now) you can't avoid a little programming.

All is not lost! The function you're about to write is very simple, it deals mainly with changing some assumptions PAWS makes about your game.

Here's the function you need to write, stripped of comments. The line numbers are *not* part of the code, don't put them in your own program, please.

```
1.      def TQUserSetUpGame():
2.          """Code Required For User Game Set up"""
3.          Global.Production = FALSE
4.          Global.Player.StartingLocation = StartCliff
5.          Global.CurrentActor = Global.Player

6.      Engine.UserSetUpGame = TQUserSetUpGame
```

OK, let's take this by the numbers, because you're going to want to study our example one line at a time. The words in **boldface** in our example are what you create, everything else is either part of the Python language or the PAWS library.

1. The first line begins the function. Make sure you start it at the very first column, because you're going to indent the rest of the function under it, just like in the example. Python always starts a function with the word **def** (short for define). *TQUserSetUpGame* (notice the capitalization) is the name you give to your function. We recommend naming your function like we did ours, the *Colossal Cave* adventure might use *CCUserSetUpGame* for example. Notice you end your function with two parentheses and a colon.
2. Line 2 is optional, it's a *comment string*. It serves two purposes. First, it gives a quick comment to human readers about what the function is and what it does. Second, Python has a special "documentation" program that can read the comment string for each function in your program and print out a sort of "program guide" for people studying your program. It's always a good idea to put comment strings on functions and methods (We'll cover methods later).
3. This line tells PAWS that your game is still "under construction", it lets you use the built-in debugging system to help you find problems in your game. When you release the game to the public you'll change the FALSE to TRUE. That will disable the debugging system, preventing knowledgeable players from using the debugging system to "cheat".
4. This line tells PAWS what room the player's character will start in. In this case, notice the location is called *StartCliff*. This is the name of the room the player starts the game in. *StartCliff* is the name of the starting room in *Thief's Quest* but you can name your starting room anything you like. In Zork, for example you might call your starting room *FrontOfHouse*.
5. The last line of the function is always the same, it sets the current actor to the player. You can copy this line between games without changing it at all.
6. Line 6 is *not* part of the function! You can tell the function ends at line 5 because line 6 isn't indented. Only lines indented under the def statement are part of the function. This line tells PAWS you're replacing the default user set up function with the one you just wrote. Notice we did *not* add parentheses to the end of *TQUserSetUpGame*. We'll explain why below.

The Devil's In The Details...

As you study our example function you'll notice we mentioned two new objects you haven't seen before, *Global* and *Engine*. Astute readers will also have noticed line 4 seems to break the rules about how to refer to properties of objects.

The Global Object

We'll look more closely at *Global* later, but the short version is that *Global* holds all the properties we want to keep handy and that don't really fit anywhere else. For instance the *Global* object holds the current screen row and column, various bits of decoded parser information, the current turn number, the player's score, and so forth.

In other words it's a big grab bag of useful information that really doesn't belong to any other object. So we created the *Global* object to give us a handy bucket to toss it all in.

Global is created by PAWS and upgraded by the Universe library.

The Engine Object

The *Engine* object is just that, it's the core game engine that powers everything else. The *Engine* object contains the parser and various pre-written routines that are the bedrock for your game. We call it an engine because it's like a car's engine—without it your game wouldn't move!

Engine is created by PAWS.

Referring To Objects

There are two ways to reference an object. The first is called *direct reference* and the second is called *indirect reference*. Both are equally important when writing your game. Each is used for a specific reason, you need them both.

Direct Reference

Direct references are simple. You use the name of the object, a dot connector, and the property you want. So in our example function you *could* have written:

```
TQMe.StartingLocation = StartCliff
```

It would have worked. (*TQMe* is the player's character). But this isn't the most flexible way.

Indirect Reference

Indirect references mean that instead of using the name of the object, you use an expression that evaluates to the name of the object instead. For example, instead of the direct approach used above we could say:

```
Global.Player = TQMe
Global.Player.StartingLocation = StartCliff
```

It does exactly the same thing as our direct reference example. This is because Python will interpret *Global.Player*, find out it means *TQMe*, then use *TQMe.StartingLocation*. By the way, by the time this function actually is run *Global.Player* will already be set to *TQMe*, that's why we didn't have to do it in our example function. I just did it in the example above to emphasize how indirect referencing works.

Why Bother With Indirect Referencing?

You may be asking why we bothered using an indirect reference when a direct one would do just as well.

There are two reasons. First, when you write your next game you can copy and past your function and make only minor changes to it. Using *Global.Player* instead of *TQMe* on line 4 of the function means that's one less change you have to make. The more games you write, the more you'll come to appreciate not having to make changes (trust me!).

Second, it conditions you to always use the phrase *Global.Player* in place of the actual player object. *Global.Player* is more generic than *TQMe*, it can apply, for example, to the *current* player's character in a game where you might allow the player to switch viewpoints from one character to another.

It also lets you create objects that refer to the player and can be easily transported from one game to another *without changing anything*. This is the Holy Grail of programming—reusability. Indirect referencing is the primary means of accomplishing that.

Your Function As Object

Did you know you can treat your functions as objects too? That's exactly what we did in line 6 of our example. Notice we set *Engine.UserSetUpGame* to *TQUserSetUpGame*. Notice we didn't end *TQUserSetUpGame* with any parentheses?

To explain why you should understand that a function can be referred to in two different ways. These are called the *evaluated* and *address* (or *object*) forms.

Evaluated Form

This is no great mystery. For example if we said:

```
Sentence = SCase("this will appear properly capitalized.")
```

Then we would have used the evaluated form of the function *SCase()*. In other words, it would take the string we put in parentheses after *SCase* and set *Sentence* equal to the string with the first letter capitalized.

This is the way *every* computer language uses functions. The evaluated form is just the result of the function. You've probably used the evaluated form of functions before and never knew what they were called.

Well, now you do. ☺

Address (Or Object) Form

This one gets a little tricky. Python stores functions at a given memory address, which is basically an area of memory referred to by a unique number, just like a house address. The same is true for objects. In fact, to Python a function *is* an object.

This means you can refer to an object indirectly. Does this sound familiar? It should, see the section *Indirect referencing* above if you skipped it.

You may see where this is going. Since a function is just an object, and you can refer to an object indirectly, that means you can *also* refer to a function indirectly. In other words, you can do *this:*

```
Engine.UserSetUpGame = TQUserSetUpGame
Engine.UserSetUpGame()
```

When this code is run, you'll actually run *your* function in place of the original! (Which, if you care, was called *default_UserSetUpGame*, it does absolutely nothing). In effect you've created an "alias" or a "synonym" for your function.

This is exactly the same idea as indirectly referencing an object we talked about above. If you're feeling brave, take a look in the *Universe.py* file at the *UniverseSetUpGame()* function. The second to last line of code is identical to the second line of code in the example above, *Engine.UserSetUpGame().*

To make a long and involved explanation short and sweet: Using indirect function references allow you to write code that's "game independent". That's exactly how Universe was written so it would work with anyone's game, that's how PAWS was written so it would work with any PAWS library, not just Universe.

Using indirect references is one of the central concepts of the Programmer's Holy Grail. Learn it, use it, live it! ☺

What We've Covered In This Chapter

This was a long chapter that covered a lot of ground. You learned how to create a function (*TQUserSetUpGame*). You learned the differences between direct and indirect referencing of objects (and functions!) and how important indirect referencing is. You also learned a bit about the *Global* and *Engine* objects, and you learned how to store a function reference so Python would treat it like an object.

Congratulations, take a break. You've earned it!

Chapter 9
Classes–Object Blueprints

In the last chapter you learned to create a function and use a few pre-defined objects. In *this* chapter you'll learn how to create the blueprints that objects are built from.

These blueprints are called *classes*.

Classes From PAWS And Universe

We already define a number of classes for you, both in the PAWS core game system and the Universe library extension. The classes in PAWS are really low level "brick and mortar" type classes. In other words, they're the building blocks of the classes you'll actually use. Just for your entertainment here are the classes from PAWS and what they're used for—you'll probably never use them directly. But do notice one thing—every class starts with the word "Class" in its name. Python doesn't demand this (you can name a class anything you like) but it's ***exceptionally helpful*** if you name your classes starting with the word "Class". This lets you distinguish between the blueprint (class) and the thing built from it (object).

Class	Purpose
ClassFundamental	This class is used to provide some of the "plumbing" required in all classes. All classes descend from one, it's the "root of all classes". At the present time it implements the *MakeCurrent()* and *Get()* metho well as the ancestor of the *SetMyProperties()* methods. Don't worry if you don't know what these me do, they're mainly used by PAWS and Universe, and you probably won't need them directly.
ClassParserError	This class is used to store messages the parser displays when an error occurs.
ClassGlobal	This class is used to create the *Global* object (remember that?). Basically this class defines properties o has no methods. (And we *will* get to methods very soon, I promise!)
ClassEngine	This class defines the *Engine* object. It is arguably the most complex class in the entire system, it form game's core functionality.
ClassBaseObject	This class is used to create all "thing" classes. It basically adds the current object's name(s) and adjectiv the parser dictionaries. It also provides minimal functionality to support the parser. In other words, it do the hard work needed when you create a new object.
ClassVerbObject	This class is used to create all verb classes. It basically adds the current verb's name(s) ("go", "walk", e the dictionary, along with the appropriate prepositions. It also handles the bulk of disambiguation fo parser. If that sounds scary, don't worry about it. For the most part you'll never have to worry a disambiguation, that's mainly the job of library writers—and you won't have to do that!

The classes from Universe are ones you *will* use; here's a very quick overview. We'll discuss them in more detail later.

Class	Purpose
ClassGameObject	This class is used to create the *Game* object. We covered the *Game* object in Chapter 7. Since there's one *Game* object (and it's already created for you) you won't use this class, though you may wa examine it.
ClassBasicThing	This class isn't really used to create objects, it's used as the basis for creating more specialized class "things". For example, *ClassActor*, *ClassRoom*, and *ClassDirection* are all more specialized versions o class.
ClassActor	This class is used to create people, animals, and monsters the player will either talk to or fight. This cl based on *ClassBasicThing*.
ClassRoom	This class defines rooms. (As if you couldn't guess, right?) This class is based on *ClassBasicThing*.
ClassDirection	This class is used to create directions (like north, south, etc). In PAWS directions are just as much obje say a lantern or a sword. You probably won't have to deal with this type of object directly, most reaso directions (including ones like in/out and upstream/downstream) are already defined for you. This cl based on *ClassBasicThing*. While that might seem strange, *ClassBasicThing* defines a great many "la nature" for objects, making it a natural choice.
ClassMonster	This class is based on *ClassActor*. It's used to define monsters, animals, or people that have combat abil
ClassPlayer	This class is based on *ClassMonster*. It defines the player character's object. Thus, in one fell swoop object of *ClassPlayer* automatically gains all the properties and methods of *ClassBasicThing*, *ClassA* and *ClassMonster*! Note that you can (and should) create your own player character class (we'll show how later) that will probably be based on this class.
ClassScenery	This class is used to create otherwise useless props to add atmosphere. For example, if you men hairbrush in a room's description you might use this class so when a player tries to take it you say hairbrush is worthless" or some such.
ClassItem	Items are simply objects that can be taken. They are described when a room is entered, they are the lam sword, the silver bars, etc.
ClassDoor	Used to create doors, or more accurately, one *side* of a door. Doors are basically scenery in other resp you can't take them and they generally won't be described as independent objects. You will notice tha said this class creates one *side* of a door. Thus a door between two rooms is actually *two* door objects for each room. This makes doors more complicated than might seem necessary, but the added flexibil worth it.
ClassLockableDoor	As you might expect, just like *ClassDoor* above except you can lock it. Remember, just like *ClassDoo* class creates only one *side* of a door.
ClassUnderHider	This one takes some explaining. It's basically *ClassItem* above, except when taken it drops its conte such a way that it looks to the player like taking the object *revealed* objects hidden under it. For exa taking a rock might reveal a scorpion under it. This is actually something of a conjuring trick, the scorpion was really *inside* the rock (store *Rock.Contents*) but the *Take* method's been altered to make it look like the scorpion was really unde rock all the time.
ClassBehindHiderItem	Just like *ClassUnderHider* above, but the item's contents appear to have been hidden *behind* self rather under it.
ClassActivatableItem	Generally used for light sources like lamps or candles or flashlights, this class lets you easily create any the player can take *and* turn on and off. It's flexible enough to support using one object to activate it lighting a candle with a match) and a second (optional) one for deactivating it. It may also easily be cha to support any on/off object, not just light sources.

Thief's Quest—A Quick Background

Before we start exploring the classes created specifically for *Thief's Quest* we should give you a quick overview of the game. The premise of the game is quite simple. You're a thief and you were caught looting a wizard's tower by the wizard himself.

As punishment he magically transported you to the site where the game takes place, a forested rift valley surrounded on all sides by sheer cliffs. Your goal is quite simple—*get out!*

Now, our setting suggests several things are likely to be quite common in the game. A rift valley surrounded by cliffs will have a lot of rooms where the player can interact with the cliffs. Also, the presence of a forest suggests there will be lots of trees. Likewise there are likely to be tree-type things, leaves, bark, and so on.

Trails are likely to be common, and there will probably be wild animals to contend with, and vast stretches of trackless forest as well.

Python is an object oriented language, meaning (among other things) that you can inherit abilities of a class by basing a new class on an existing one. Further, you can layer this as deeply as you need to, even with your own classes. We take advantage of this inheritance ability in Thief's Quest to perform all sorts of labor-saving programming tricks.

Thief's Quest Foundation Classes

Remember, as a game author you don't want to work any harder than you absolutely have to. This means you have to work *smart*, to avoid drudgery. So the first thing to do is consider how to re-use the things you write in different situations. In PAWS this usually means creating *classes.*

Let's take our example from *Thief's Quest.*

The Anatomy Of A Class

Look at *ClassTQRoom* in the game. There's a short comment explaining what the class is intended for. Basically, our game has certain properties that are common to all rooms. These properties aren't supplied by the Universe library, nor by the PAWS engine, so we'll have to create them ourselves.

In *TQ* we're going to need several kinds of rooms, some for outside, some for caverns, and so on. Every one of these kinds of rooms will need the same properties. The smart way to do this is to create a base class, set them there, then use that base class to create all the other classes (blueprints, remember?) we need.

Defining The Class

The first statement starts the class definition:

```
class[1] ClassTQRoom[2](ServiceDictDescription,ClassRoom[3]):
```

1. The word *class* signals the beginning of a class definition (a class is just a blueprint, remember?) Notice how it's at the left edge of the page? Everything that is part of the class definition will be indented under this line.
2. *ClassTQRoom* is the name of the class you're creating.
3. Inside the parentheses following the name of the class you place a comma separated list of classes you're basing your class on. In this case there are two, *ServiceDictDescription* and *ClassRoom. ServiceDictDescription* is a type of class we haven't talked about yet, a *service.* A service is sort of a "mini" class, it's used to add abilities to classes. A service isn't a full blown class, in this case the service allows you to more easily define descriptions for rooms. *ClassRoom* you should recognize from the table above, this class lets you define object that do "room" sort of things, like describe themselves and allow the player to enter them. Please note that you *always* list services before the class, Python may ignore the service if you list it after the class. Always list the service first!

Class Properties

Here's a tricky concept, the *class property.* A class property is just like the object properties you've already learned about, they store a number, a string, a list, a dictionary, or whatever. In *ClassTQRoom*, the *DefaultMap* property is a class property, not an object property. You can tell this because it doesn't begin with "*self.*".

The difference is that class properties are *shared* between all objects created with that class. For example, no matter how many rooms you create from this class they will all share the same *DefaultMap* dictionary. Let's say you create *Room1*, *Room2*, and *Room3* from this class.

How many copies of *DefaultMap* are there? Just one. Unlike, for instance, *HasStream* (3 copies, one per room), there's only *one* copy of *DefaultMap* between the 3 rooms.

So why use class properties? Two reasons, first, of course, they save memory. This isn't as important as it used to be, but it's still a good idea to conserve as much as you can.

Second, it gives you a single copy in case you want to change one property and have it affect all rooms. This might be handy, for example, if you've created a set of jail cells from a class with a class property that controls whether the cell doors are open or closed. (They're all open or they're all closed).

Methods

Told you we'd get around to explaining methods. A method is nothing more than a function you define *inside* a class definition!

This makes the method (function) name unique to that class. For example, both *ClassBasicThing* and *ClassRoom* define the method *Enter()*, but Python knows that *ClassBasicThing.Enter()* and *ClassRoom.Enter()* are two different methods (Remember a method is just a function defined inside a particular class). The methods have the same name but act differently.

A method also has one small peculiarity that makes it different from a function. There's an extra, invisible argument. You only use it in two places. First, when you define a method you put it in the parentheses after the method name (see the examples in the *TQClassRoom* class). When done this way, always call the argument *self*, this is a Python convention we recommend you follow religiously.

The second (rare) time you use it is if you call a method directly from an ancestor *class*, as opposed to objects created from the class.

For example, let's say you created a room called *Room1* from *TQClassRoom*. And let's further say that you wanted to call the *FeelDesc()* method from each. Here's how you'd do it:

```
Room1.FeelDesc()
ClassRoom.FeelDesc(Room1)
```

In case you're shaking your head in bewilderment, don't worry. You won't need to call a method from a class very often, usually you do it from the object. We'll cover this in the *SetMyProperties* and *FeelDesc* methods below.

The SetMyProperties() method

In each and every class you define you'll create a method called *SetMyProperties()*, with an argument of *self*. This method lets you create object properties when the object is created from the class. For instance, if you said:

```
Room1 = ClassTQRoom()
```

Python will actually do this:

```
ClassTQRoom.SetMyProperties(Room1)
```

What Python actually does is much more complex, but unless you need to create a library you don't have to worry about it. If you *are* creating a library, look at *ClassBaseObject* in *PAWS.py* to see what else is happening behind the scenes.

In *ClassTQRoom* you'll notice the first line is:

```
ClassRoom.SetMyProperties(self)
```

This is an example of the rare time you actually use a method's invisible argument. If you remember, *ClassTQRoom* is based on the class from Universe called *ClassRoom*. What we are doing here is calling the *SetMyProperties* method of *ClassRoom*. This will add all the properties from *ClassRoom* to our object before we add any new ones.

In case you hadn't guessed by now, *self* is a variable that holds the object being created. In our *Room1* example, *self* would be *Room1*. So in effect, the statement:

```
Room1 = ClassTQRoom()
```

Actually translates to:

```
[_some code you can ignore_]
ClassTQRoom.SetMyProperties(Room1)
ClassRoom.SetMyProperties(Room1)
```

Whenever you create a class you *always* define a *SetMyProperties* method and call the ancestor class's *SetMyProperties* method just as in this example. This is how properties get inherited from one class to another. Methods, on the other hand, are inherited automatically, you don't have to do anything.

If you look at the full text of *SetMyProperties* from *ClassTQRoom* it looks like this:

```
def SetMyProperties(self):
    ClassRoom.SetMyProperties(self)
    ServiceDictDescription.SetServiceProperties(self)
    self.CliffMessageIsVisible = FALSE
    self.HasBeechTree = FALSE
    self.HasBirchTree = FALSE
    self.HasCliff = FALSE
    self.HasForest = FALSE
    self.HasForestPath = FALSE
    self.HasMapleTree = FALSE
    self.HasMound = FALSE
    self.HasPineTree = FALSE
    self.HasStream = FALSE
    self.IsOutside = TRUE
```

The first line we've already explained, it calls *SetMyProperties* from the class you're basing *ClassTQRoom* on. That adds all the inherited properties to our object. The second line does the same thing for any properties needed by the *ServiceDictDescription* service. A service is nothing more than a "mini" class. Services provide special abilities that are always self contained and never conflict with a class. Services have no ancestors and you don't use them to base other services on. Services are used to provide a *small*, well-defined set of functions to a class. Services are deliberately designed to be usable with any class, to provide the ultimate in flexibility.

The remaining lines set up our new properties, like *HasForest* or *IsOutside*. Notice how "*self.*" begins each property? Remember that when the object is created, *self* is an indirect reference to the room being created. Thus if we were creating *Room1*, then *self* would hold *Room1* and the text would actually say *Room1.IsOutside = TRUE*.

The FeelDesc() method

To understand why you need to redefine this method (which returns a string holding the appropriate description to "feel") you have to understand that the *ServiceDictDescription* service handles both rooms and things. The *FeelDesc()* method in the service doesn't give an appropriate response, therefore we have to override the inherited method, which we do simply by defining a new one.

You'll notice it's a very short definition, only one line:

```
def FeelDesc(self): return ClassRoom.FeelDesc(self)
```

This is really a clever bit of leveraging existing code. What it's doing is returning the string that calling *ClassRoom.FeelDesc()* generates. Let's assume the player is in *Room1* and types "Feel around". The parser (after much crunchings and munchings) will call *Room1.FeelDesc()* to display a response.

Which in turn calls *ClassRoom.FeelDesc(Room1)*. Which gives a reasonable response to any room you might create, all without you having to worry about it!

This is yet another example of how indirect referencing can make your life simpler—at least in the long run. I'm sure you're feeling a bit punch drunk right now, so let's take a break.

What We've Covered In This Chapter

This was another long chapter that covered a lot of ground. If you've managed to survive so far, congratulations! You're at the top of the mountain, you've gained all the secret knowledge of the gurus about classes. From now on it's all down hill.

In this chapter you discovered how to create classes (blueprints) of the objects you'll use throughout your game. You encountered all the basic classes in Universe, you discovered the anatomy of a class, and that every class needs to have a *SetMyProperties()* method. You even discovered the deep dark mystery of how objects inherit their properties.

Finally you discovered the difference between a *class* property and an *object* property. There's only one copy of a class property, it's shared between all objects created by your class. Each object, however, has its own copies of object properties.

As a bonus you also learned how to call ancestor methods, methods that exist in the class you're basing your class on. And the cherry on top, you discovered a new kind of miniature class called a *service*, that provides a basic class some extra abilities.

Chapter 10
The Great Outdoors!

Since the last two chapters forced you to leap tall concepts with a single bound, this chapter will take it easy on you. All we'll do is talk about a new foundation class, called *ClassOutside*.

This class is the basis for all rooms that are "outside", as opposed to "inside". Just about all the rooms on the surface count as outside rooms, except for one or two small caves.

The Purpose of This Class

ClassOutside was created to set up reasonable defaults for rooms that are in the outdoors. If you think about it, what kind of changes would you make to *ClassTQRoom*? That's what we're basing *ClassOutside* on.

You'll remember that *ClassTQRoom* was designed to set all the new room properties we needed in *Thief's Quest* that weren't already available in Universe or PAWS.

As it turns out, the only real change we need to make is to create a new *DefaultMap* class property.

A Map? No, It's A Dictionary!

DefaultMap is a dictionary. In Python a dictionary is a special variable type. Like any normal dictionary it has *keys* and a *values*. Dictionaries can contain as many entries as you like. Here's the *DefaultMap* for *ClassOutside*.

```
DefaultMap = {North:       "You can't go that way.",
              Northeast:   "You can't go that way.",
              East:        "You can't go that way.",
              Southeast:   "You can't go that way.",
              South:       "You can't go that way.",
              Southwest:   "You can't go that way.",
              West:        "You can't go that way.",
              Northwest:   "You can't go that way.",
              Up:          "There's nothing climbable here.",
              Down:        "There's no way down.",
              Upstream:    "There's no stream here.",
              Downstream:  "There's no stream here.",
              In:          "There's nothing here to enter.",
              Out:         "There's nothing here to exit."}
```

Ok, let's notice some things about *DefaultMap*. First, notice that it looks like we're setting *DefaultMap* to something, notice how the first line says "*DefaultMap* ="?

We are. Notice that the entire definition is surrounded by curly braces, the symbols "{" and "}". This marks the beginning and end of the dictionary definition.

Second, notice how we put each key and value on its own line. This isn't strictly necessary, but it does make the code appear much easier to read, and that *is* important. Notice also that we've spaced the strings so that they begin in the same column. Again, this is to make the code easier to read, it really doesn't matter as long as there's at least one space between the colon and the value. Also notice that each key/value is separated by a comma.

Each line has a key, then a colon character ":", then a string. The string is the value. So, for instance the first key is *North* (a direction object, by the way), and the value is "You can't go that way".

What The *DefaultMap* Is Used For

Ok, you know how to create the *DefaultMap*, but you have only a vague idea as to *why*. The reason is to make your life simpler when creating a map of your game.

Let's assume the player is walking through a narrow cleft. He can basically go north or south. If he tries to go any other direction the game should complain with "You can't go that way."

There are 14 basic directions the player can go. When setting up a map, do you *really* want to have to have write up to 12 "You can't go that way" strings *for every room*, when you can write them just once?

That's what *DefaultMap* is for. If you create a map for a given room you only have to include the directions that actually lead to another room, or provide a special complaint. If the player tries to go in a direction that isn't in the room's map then the *DefaultMap* is consulted for the appropriate direction.

This results in a huge savings in memory, disk, and typing!

The Obligatory *SetMyProperties()* method

You'll notice that our *SetMyProperties()* method for this class is almost empty—the only line in it calls the *SetMyProperties()* method from *ClassTQRoom*. Even though this class adds no properties of its own you still need this method because you may later base another class on *this* one. (*ClassOutside* is based on *ClassTQRoom*, remember, and we created *ClassTQRoom* too).

What We've Covered In This Chapter

Told you this was going to be an easy chapter. In this chapter you learned what *ClassOutside* is intended for, what dictionaries are, and why *DefaultMap* is an important class property.

ClassOutside is a very important class, you'll base a variety of other classes on it, like *ClassCliffRoom* and *ClassForest*.

Chapter 11
Caverns

We've got one class for outside, since there's lots of outside in the game. But since *Thief's Quest* is also a dungeon crawl (meaning a cave exploration with monsters) we also have lots of caverns, and a class to make them easier to make.

ClassCavern is pretty similar in concept to *ClassOutside*. We create a *DefaultMap* and fill it with a slightly different complaint ("There's a wall there.") than we used for the outside.

Like *ClassOutside*, *ClassCavern* is based on *ClassTQRoom*. However, since caves are dark the *SetMyProperties()* method contains a line that reads:

```
self.IsLit = FALSE
```

This makes the room dark, and PAWS will treat it appropriately (not describing it unless the player's carrying a light source, etc). Notice with one line of code we profoundly affect the room's behavior, and do it in a way that makes perfect sense. (In other words, *you* don't have to do any extra work!)

Notice the next line:

```
self.NamePhrase = "Cavern"
```

This sets the "name" of the room (used when the player needs to refer to the room, such as when teleporting with the magic staff, etc). The room's name is always displayed first, just before the long description of the room is printed. For example:

```
Waiting Room

You are in a small non-descript room with beat up furniture and dull grey
walls. The only exit is guarded by a sign saying "WAIT HERE".
```

Every room has a *NamePhrase*, no exceptions. The reason we set this property in the class rather than waiting is to make your life a little simpler, you don't *have* to name your caverns to make them work.

And finally we have the two lines:

```
        self.SetDesc("Odor","The air is very fresh, but carries no odors.")
        self.SetDesc("Sound","I can't hear anything.  The silence is
deafening.")
```

These two lines set the default odor description and sound description of all rooms defined with this class. That way you don't have to change the descriptions unless there's something special about the room. Most caverns are silent, and have fresh air that don't carry many odors.

Notice the *SetDesc()* method is used to create the descriptions. *SetDesc()* comes from the *ServiceDictDescription* service (remember a service is just a "mini" class). It puts all descriptions inside a dictionary for easy manipulation. All sensory descriptions are handled this way.

Lit Caverns

As you might guess, there's a special class for lit caverns as well as dark caverns, called *ClassLitCavern*, based on *ClassCavern*. If you examine the code you'll notice that *ClassLitCavern* is pretty short. It has no *DefaultMap* of its own (it inherits the one from *ClassCavern*). It *does* have a *SetMyProperties()* method (all classes must have this method!) which changes one property, *self.IsLit = TRUE*.

Everything else about the class is inherited from *ClassCavern*. Now you begin to see the benefits of all this object oriented nonsense.

What We've Covered In This Chapter

This was another easy chapter. We talked about how you make a room dark, the importance of *NamePhrase*, and introduced you to *SetDesc()*.

ClassCavern is used to make cavern rooms directly, it also is the parent (ancestor) of *ClassLitCavern* for when you want a cave that is lit up.

Chapter 12
Other Room Classes

This chapter covers several classes. All are pretty simple, we've done most of the work by defining the ancestor classes covered in the previous chapters.

Outdoor Cliffs

The next class we want to examine defines rooms that not only are outdoors, but also have a cliff present. Since the game takes place in a valley surrounded by cliffs, there are a lot of rooms that have a cliff that the player can interact with.

Notice *ClassCliffRoom* isn't very complex. It has a *DefaultMap* to give appropriate messages if the player tries to move in a direction blocked by the cliff, and it has one property, *HasCliff* set to TRUE.

You might think this is a lot of effort for setting one property, but remember all rooms created from this class will share a single copy of *DefaultMap*. That alone makes it worth the effort.

Valley Walls

Some rooms (like the *StartCliff* room) have a cliff in them, but have lots of other interesting things to distract the player. Other rooms, specifically the north and south valley walls, are intended to reinforce the feeling of being trapped by the cliffs, so there isn't much else *except* the cliff in the room.

These rooms also have pretty much the same descriptions too. A perfect opportunity to create classes to leverage the commonalities between the dozen or so rooms.

ClassValleyWall

This class is based (not surprisingly) on *ClassCliffRoom* and holds everything that's common between the north and south valley walls. This includes the *DefaultMap*, the odor description, and the sound description. We won't actually create any rooms with this class, instead, like so many other classes discussed so far we'll use it to create more classes with.

ClassNorthValleyWall

This class is based on *ClassValleyWall* and adds the *NamePhrase* shared by all rooms created with this class, and it sets the long description (also common to all rooms of this class).

ClassSouthValleyWall

This class is identical to *ClassNorthValleyWall*, except of course the *NamePhrase* and long description have been changed appropriately.

Forests

The other thing our game's valley has a lot of are forests. Some parts are trackless, and easy to get lost in. Other parts have paths. Thus we have two classes to handle forests.

ClassForest

This class is similar to the ones we've covered already. It's based on *ClassOutside*, has its own *DefaultMap*, and sets the *NamePhrase*, plus long, odor, and sound descriptions. It also sets the properties *HasForest* and *HasPineTree* to TRUE.

This class is used to create rooms that the wolf can wander through. Most rooms created with this class are meant to be confusing masses of trees where it's very hard to find one's way. This makes the wolf particularly dangerous…

ClassForestPath

This simple class is based on *ClassForest*, inheriting everything from that class. It also sets *HasForestPath* to TRUE. This bars the wolf from entering the room.

What We've Covered In This Chapter

This was a really short easy chapter. You encountered the last of the classes used to create rooms. You saw that certain areas of the game have a lot of rooms that are of the same type, such as the north and south valley walls, the forest, and so on.

You also learned that when you have commonalities you should create a class to take advantage of them. Not only does this free you from extra typing (always a good thing!) it also lets you make your game smaller without ill effect. This too is a good thing.

Chapter 13
Landmarks—How To Create "Floating" Scenery

There is a Universe class we haven't dealt with yet, it's the *ClassLandmark* class. We're dealing with it now because it's a "foundation" class, but one that comes with Universe rather than one I created in *Thief's Quest*.

This class is your first introduction to floating objects, in particular floating scenery. Floating objects, as their name implies, don't have a fixed location. Instead they either return "None" as their location, or the player's location when the correct circumstances exist.

Floating scenery is incredibly useful any time you want a piece of scenery to appear in different rooms. The secret of any floating object class (not just *ClassLandmark*) is that the *Where()* method has been modified. Instead of returning *Object.Location* it will return either *None* (if the object shouldn't appear in the room) or *Global.CurrentActor.Where()* if the player can see it. Remember if you describe a tree in a room the player may want to touch it, take it, sniff it, etc. Having a single scenery object (like a tree) that can appear in any forest room means you only have to code one tree, and thus you can afford to make the tree more elaborate than if you had to code a dozen different trees. Without floating scenery, and even with inheritance you would still have the extra memory consumption to no good purpose.

In TQ there are several objects that use *ClassLandmark*. The cliff, the pine tree (and needles and cones and resin), the mound, etc.

Smoke And Mirrors

This class is challenging only because it introduces a new technique, the *Get()* method. Otherwise it's a standard class based on *ClassScenery*. First, notice there's a new property, called *Landmark*, defined as an empty string.

Let's take a look at the *Where()* method defined for this class:

```
def Where(self):
     if Global.CurrentActor.Where().Get("Has" + self.Landmark):
          return Global.CurrentActor.Where()
     else:
          return None
```

What is that crazy IF test doing? Ok, break it down. *Global.CurrentActor* we've seen before, it's an indirect reference to the player's character. In *Thief's Quest*, this would be *TQMe*.

So *TQMe.Where()* translates to the player's location, in other words, the room the player is currently in. Let's say that was *StartCliff*. Just another indirect reference, you've seen them before, right?

Our huge unwieldy IF test is beginning to make more sense. So far we've translated *Global.CurrentActor.Where().Get()* to *StartCliff.Get()*.

The *Get()* method is a really neat programming trick. It takes the name of a property or method as a *string* argument, and returns the value of the property or the value returned by the method.

Let's assume we've made an object called *PineTree* from *ClassLandmark*, and set *PineTree.Landmark* property to "PineTree". Then "Has" + *self.Landmark* translates to "HasPineTree". Thus our IF test above actually says:

```
if StartCliff.Get("HasPineTree"):
```

Which can further be refined to:

```
if StartCliff.HasPineTree:
```

Why go to all the pain and anguish of that huge long if test? Because of indirect referencing. What appears to be verbose and unwieldy is actually a way to make the if test work for *anything* defined with *ClassLandmark*, which also would include things like leaves, pine cones, tree bark, etc. All you need to do is define the particular *Landmark* so that it assembles to the proper *Has** format (*HasPineTree*, *HasCliff*, *HasStream*, etc.), make sure all rooms have the particular *Has** property (which you can add to the *SetMyProperties()* method of *ClassTQRoom*) and you can add as many tree types as you like.

You can use this technique in your own games as well. This technique works well when you have something generic (like trees) that have individual differences (like pine, beech, etc). Automobiles, for instance. (Ford, Chevy, etc).

Notice you don't *have* to define a *Has* method for every room, the *Get()* method will return *None* when a property doesn't exist. But this can slow your game down on slower computers (such as handhelds). Of course you also have to balance the added memory requirements against the added speed...

What We've Covered In This Chapter

We really only introduced one new technique (the *Get()* method) and gave you further details on the *Where()* method and how you can use it to create floating objects. If you bothered to look at *ClassBasicThing* you'd see that the standard *Where()* method is defined like this:

```
def Where(self):
     return self.Location
```

In other words, it merely returns the object's *Location* property. Our more complex version in this class allows the scenery to float around with the player, only appearing when the correct property is TRUE.

Chapter 14
The Ground And The Sky

You may not even have thought about these. After all, who spends time in a game examining the ground or the sky? However, in certain games (particularly mysteries) examining the ground for minute clues may be important. Likewise in other games the condition of the sky (as in threatening weather) might be vital to a player's survival.

On the other hand, most of the time you (and the player) won't really care about the ground (which is also the floor) or sky (which is also the ceiling). To handle both situations Universe created a pair of generic objects for you.

The Ground

The Ground object is created from ClassLandmark, its Landmark property is Ground. This means Ground will be present in any room with a *HasGround* property set to TRUE, which by default are all rooms. For everything else the *HasGround* property is set to FALSE. Remember, the Ground object also represents floors when the player is inside a building.

The only really interesting thing about Ground is its *LDesc* property. If the player types "Examine Ground" the ground object will give the *room's GroundDesc* property instead. This allows you to easily change the ground description for each room without having to create new ground objects for each room.

All the other sensory descriptions say "It sounds like ordinary ground to me" or "It feels like ordinary ground to me". We'll talk about how to create custom ground for the senses other than sight in a moment.

The Sky

The Sky object is created from ClassLandmark, its Landmark property is Sky. This means Sky will be present in any room with a *HasSky* property set to TRUE, which by default are all rooms. For everything else the *HasSky* property is set to FALSE. Remember the Sky object also represents the ceiling when the player is indoors.

The Sky object is one that most games won't change, so the default properties become more important. The sound and odor properties return the *room's* properties, not the Sky object's. The LDesc property defaults to the *room's* SkyDesc() method.

Creating Custom Ground And Sky

There are two levels of customization available for the ground and sky objects. First, you can use the *room's GroundDesc()* and *SkyDesc()* methods to give a room's ground (floor) and sky (ceiling) sight descriptions. This is the easiest method, both the *ClassRoom* class and *ServiceDictDescription* service support this right out of the box. The argument for *SetDesc()* is "Ground" and "Sky" respectively.

Full Ground And Sky Customizations

The following discussion applies equally to creating custom Sky as well as Ground for a room, but you'll almost never find a need to customize the Sky object because it already supports smell and sound, and rarely will a player touch the sky (ceiling) of a room.

Sometimes you want a full sensory experience for the Ground object, particularly the senses of smell, hearing, and touch. To achieve this you need to create a *new* custom ground object. This object can be created with either *ClassScenery* (for ground/floor unique to a certain room) or you can use *ClassLandmark* to create ground that floats to several different rooms.

In either case you need to set the room's *HasGround* property to FALSE. This will make sure the regular Ground object doesn't appear. If you're using ClassLandmark you should set your landmark property to TRUE.

Then create your new object and set the sensory descriptions like you would for either scenery or landmark objects.

Let's take an example. In TQ there's a lot of forest, and we created a *ClassLandmark* object called *ForestFloor* to give a full sensory description of the forest. As luck would have it we already have a landmark property we can use, *HasPineTree*. This property is used for several landmark objects (*PineTree, PineForest, PineCone, etc*). So all we have to do is set *HasGround* to FALSE, create *ForestFloor*, and voila! We're done.

The *ForestFloor* object is just a standard *ClassLandmark* object. One small trick, we turned off the *HasGround* property in the foundation class *ClassForest*, as well as in other forested rooms that weren't created with *ClassForest*. If we didn't PAWS would print the description for *both* ground objects!

Chapter 15
Walls And No Walls

We've defined a class landmark object for you called *Wall*. Like *Ground* and *Sky*, the ***Wall*** object uses the *room's WallDesc* property to define its look definition and has default definitions for the other senses. By default the *HasWall* property is set to TRUE in the *ClassRoom* class, so by default all rooms have walls.

As you might have guessed there's a wrinkle. What happens to rooms that don't have walls? (Like anyplace in the outdoors).

Simply set the *HasWall* property to false for that room. That will make the *NoWall* object appear. The *NoWall* object has sensory descriptions that basically tell the player there is no wall. For example "Examine Wall" will say "There's no wall here.", which is *NoWall's* long description!

NoWall is created using the *ClassLandmarkMissing* class, which is identical to the ***ClassLandmark*** class, except the object appears when the landmark property is *false*. In other words, *NoWall* appears when *HasWall* is false.

Customizing Walls

Unlike the *Ground* and *Sky* objects, where all you have to do is turn off the ***HasGround*** or ***Has Sky*** properties, turning off ***HasWall*** doesn't just make the wall disappear, it makes *NoWall* appear. This makes creating your own wall impossible because your wall and the *NoWall* description would appear together, which would make no sense to your player.

Fortunately you can get around this problem by setting the *ParserFavors* property of your custom wall to TRUE. ***ParserFavors*** literally forces the parser to choose your object when two identical objects appear in the same room. At the present time only *NoWall* will appear in areas you might want to use your own wall object, but in the future other such conflicts may arise as well.

For an example of a custom wall see the *Cliff* object in Thief's Quest.

Chapter 16
A Retrospective And A Look Ahead

We've covered an awful lot of ground so far. It's time to take a break, because when we continue with this chapter we're going to show you just how much you've learned already and give you some tips on how to proceed. So put the book down, go get some hot chocolate, and watch TV for a while, or whatever helps you relax.

When you come back be ready for a test!

A Review

The steps of creating a game are still the same. First, complete the game in your head. Understand how all the rooms fit together in a map, how the actors will interact with the player (and each other), what the plot of the game is, what major tasks the player has to perform, and so on.

The next step is to begin to lay out your rooms and look at them in terms of common features. For example, in *Thief's Quest* all caverns are dark, all outside rooms have light. We know that the valley walls look pretty much alike, and that forest rooms are pretty much the same whether or not they have paths.

Doing this allows you to design your class *hierarchy*, meaning which classes inherit what from whom. Always remember a good class hierarchy will cut days or weeks (or even months!) from the time it takes you to create your game.

A good class hierarchy is important, but at the same time it's tricky to get right. Invest the time now and you'll reap huge rewards later, both in time saved and programming simplicity. I can't stress this enough.

Take the time to do it right or take the time to do it over. (And over, and over, and...)

Your choice.

Implementation Order

A game has 3 major pieces, the *sets*, the *props*, and the *actors*. This is the order you should implement them in.

The Sets

The sets are your rooms. Ideally, your game should have sets that are evocative. Your rooms are the backdrop of your story. They should evoke the mood of your game. Vast vistas, or dismal, foggy swamps, or a horrific dark mansion. Whatever your game is about, your rooms should reflect it, and reinforce it. Unlike graphic games you only have words to paint your mindscapes. Use them wisely.

Remember, you're creating an *interactive fiction* game. That's basically a novel the player can become directly involved in, a novel where the player's choices make a real difference. There's no substitute for good, solid, *writing*. You're creating a game, yes, but more importantly, you're creating a *story*.

I recommend that you finish all the rooms in one section (or level, if you think in terms of Dungeons & Dragons) at one time. Then create the map for those rooms and run your game. Create a minimal player character so you can wander around your creation, seeing how the rooms feel. Note any spelling errors and rough edges in descriptions. This is the time you want to fix them. Your rooms become the supporting foundation of the game, best get them right before moving on.

Also note in your wording all the *scenery*. In this context scenery is any object a room happens to mention that the player may try to interact with. The richer your descriptions, the more scenery you're going to have to deal with.

For instance, in a forest trees are scenery, leaves are scenery, the carpet of fallen needles on the ground, the ground itself, all this is scenery. Responding to a player's attempts at manipulating scenery objects can change your game from fun to mind-blowing. Do it right, and your game becomes a textual version of *Myst*, in other words, a "real" place. Clever handling of scenery maintains the player's willing suspension of disbelief.

After all, there's nothing worse than having a description mention a statue, and when the player tries to touch it or take it the game responds "I don't see a statue here", or something equally inane.

The Props

Once you've gotten the sets in place, now it's time to start creating the props. Broadly speaking props come in two varieties. The first is *scenery*, the second is *things*.

Scenery

To create a truly great game you're going to spend a tremendous portion of your development time and effort dealing with scenery. Describe a room with a rug, table, chair, and clock and I guarantee that some player will try to do the most outrageous things with objects you (thought you) were just mentioning in passing. Mention *anything* in a description, no matter how trivial, and the player is going to think it's important.

There are a lot of reasons for this. The player may be stuck, and trying to play "guess the object that gets me out of here". Or they may hare off after a wild goose, completely missing the plot thread you were trying to have them follow, utterly convinced their bizarre solution makes perfect sense. (Which it does, *to them*).

Scenery has one aspect that makes it different from things—you can't take it. There's always got to be a logical reason why you can't take it, even if it's something like "Realizing the uselessness of collecting grass, you don't bother".

Once your rooms are complete, put all the scenery you can into place. It's tedious, it takes a long time, and many players will never interact with it, but in the end it will make your game that much more "real" to the player.

Things

Things are what most people think of when they hear the word "object". A thing is an object the player can pick up and carry. As such things have a few extra properties and methods that scenery doesn't need, but we'll get to that later.

Once all your scenery is in place, create your things. Put them where you want the player to find them.

Once all the rooms, scenery, and things are done for a particular area/level, wander around and try to break your game. This is the stage where you'll spend a lot of time too. Do totally off the wall things as well as normal stuff. The more bugs you find now, the less your players will find!

Tricks And Traps And Bells And Whistles

This category contains the puzzles that are neither rooms, nor scenery, nor props. You may have a waterfall that hides a secret cave, or a chasm that the player can cross only by waving a magic wand that creates a bridge.

This is what most people think of when they think about writing a game. This can be the part of your game you enjoy writing the most, it will certainly be the part where you write the most programming code.

But do keep in mind the more complex the puzzle, the harder it will be for your players to solve. Don't make the mistake of getting so caught up in the delights of programming a clever trick or trap that you forget to step back and view the puzzle from a player's point of view.

Always step back and view the puzzle from a distance. Have you provided enough clues to let the player solve it? All too often game authors make the mistake of forgetting the player won't have encountered a single critical clue until *after* they encounter the puzzle. This is particularly harsh if the trap kills the player because they can't solve it or didn't have the clue they needed to avoid it in the first place.

Always remember the player's bill of rights. Treat the player well and they'll keep coming back to play the new games you write!

The Actors

Once you've created the rooms, placed the scenery and other props, and finished programming all the puzzles and traps—*and tested it all*, the final, and perhaps hardest step, is to create the actors.

Actors are difficult to create because they have to model living creatures, with all the (perceived) complexity of animals and even sentient beings. Actors can be conversed with, fought, or outwitted. All this requires code. And while actors have some commonalities, they don't have a lot.

Creating a good actor takes time and patience. It also takes the realization that you're going to have to cheat—a *lot*, to make the actor "real". This cheating comes in the form of cutting corners. If you ask an actor something they may not know, lie, or tell what they know. In any case, you should think about having a random selection of responses for any given question. Combat with actors should always have a large selection of hit/miss responses, which will make the combat more interesting.

Chapter 17
How To Create Rooms

You've been very patient, sitting through chapter after chapter on theory and class construction. Congratulations, you've graduated, it's time to learn how to make rooms, scenery, things, and actors. Along the way we'll also show you a few tricks/traps and how to make them.

In this chapter we'll concentrate on rooms. We'll cover the basics and a few specialized tricks. Keep in mind, the examples we'll show are from *Thief's Quest*, and many of them use the classes we defined in previous chapters. If you should become confused as to why a particular class was chosen, *stop*. Take the time to go back to the chapter about that class and re-read it. In the end this will save you hours of frustration.

Just The Facts, Ma'am

Every time you create rooms there are a few basic lines of code you type, they differ only in detail. For example, let's look at the *StartCliff* room, the first room defined in *TQ.py*.

Here's the first few lines of code, the line numbers are just for reference.

```
1. StartCliff = ClassCliffRoom()
2. StartCliff.CliffMessageIsVisible = TRUE
3. StartCliff.HasForestPath = TRUE
4. StartCliff.HasPineTree = TRUE
5. StartCliff.NamePhrase = "At 'Start'"
```

Let's examine each line individually.

1. This line creates the *StartCliff* room. Notice how it looks like you're calling *ClassCliffRoom()* as a function? That's because, in a way, it *is* a function. Behind the scenes Python is jumping through all kinds of hoops to make this work. But you don't have to worry about what's happening, or why, just how to set it in motion.
2. This is just setting a property of *StartCliff* to TRUE. As it happens, the word "Start" is carved halfway up the cliff in letters 50 feet tall. This property is TRUE in rooms where the player can see the carving. This involves only a few rooms because the forest blocks the view in most areas.
3. Another property being set to true, this keeps the wolf out of the room and allows the *ForestPath* floating scenery to appear.
4. There are pine trees in this room. Like the line above, this allows a piece of floating scenery appear.
5. *NamePhrase* is <u>very</u> important for rooms. It names the room, this name appears just before the room's long description when a player enters the room. *All* rooms must have this property set, without exception, although in some cases (like rooms created in *ClassForest*), the class automatically sets this property so you don't have to.

Sensory Descriptions

PAWS and Universe support all 5 senses without additional programming. Obviously, the sense of taste isn't applicable to rooms. You can say "listen" to listen in general, but you can't say "taste", it just doesn't make any sense.

To set the sensory descriptions of rooms defined with the *ServiceDictDescription* service (remember the miniature classes called services?) you use the *SetDesc()* method. This method uses two arguments, the first is the sense the description applies to, the second is the description itself. Here's a table showing the available senses and what they're called. Please note they must be spelled and capitalized *exactly* as they appear on the table, or they won't work.

Argument	Sense	Notes
"L"	Sight	L is actually short for Long, as in Long Description. The long description is obviously what the player sees. All text adventure games support something similar.
"Sound"	Hearing	Many games support listening, this is the second most common sense found in IF games.
"Odor"	Smell	Not many games support smelling, except for the most blatant odors, which are usually mentioned in the long description of the room.
"Feel"	Touch	This sense is overridden for rooms, it always returns something like "Scrabbling about with your hands reveals nothing".
"Taste"	Taste	This sense obviously isn't supported for rooms, but it is supported for scenery and things. *ServiceDictDescription* can be used by *any* class, not just rooms.

TQ uses the non-visual senses heavily. This allows subtle sounds, odors, and even tastes and tactile sensations to provide vital clues to the player's survival. Early in the game the player encounters what he or she believes to be a woman (actually a dryad) who gives a veiled clue concerning the importance of the non-visual senses.

In your own game you can ignore the non-visual senses if you like, there are suitable defaults built into the Universe library to handle the player's experimental "listen" or "sniff powder" commands.

Use Of Curly Brace Expressions

If you look at the sound description for *StartCliff* you'll notice a very odd piece of text surrounded by "{" and "}" characters.

This is called a "curly brace expression". It allows you to literally put any Python expression between the curly braces and have it evaluated just before the string is printed on the screen. *This expression must evaluate to a value that can be printed.*

For instance the expression {SCase(You())} will usually evaluate to the word "You" (notice the capitalization). However, the *You()* function uses *Global.CurrentActor*, if the player were to say something like "Fred, listen", the computer would evaluate *Global.CurrentActor* to be Fred instead of the player, and would respond with "Fred" instead of "You".

This is a *very* powerful ability. It makes writing generic text much easier when you can't guarantee in advance what the wording should be. The word "you" in the sound description is a prime example.

Whenever text can change given the value of *Global.CurrentActor*, it behooves you to use a curly brace expression (also known as a CBE) instead of hard-coding text or using some vague wording to cover multiple subjects.

CBE's are easy, quick, and effective. You do have to test more when using them just to cover the odd situation of "Shovel, listen". ☺

CBE's are useful with things other than *Global.CurrentActor* as well. If you ever find yourself in a situation where the text might change with circumstances CBE's are the way to go.

Overriding Room Methods

There's good news and there's bad news about overriding methods. First, the bad news. In order to override a method you have to create a new class, which means you may be creating a lot of classes that will have only one object. That isn't as bad as it sounds, since you normally won't need to override methods that often.

The good news is that creating a room specific class gives you all kinds of flexibility. You can, for instance, add completely new methods if you need to, or override any existing ones.

The room *A4WayPath* gives an example of how to do this. We need to override the standard *SoundDesc()* method for this room, because what you hear when you listen depends on whether or not the Dryad is present in the Maple Stand to the north.

As you can see, we create a class called *ClassA4WayPath*, based on *ClassForestPath* and define a *SoundDesc()* method. By the way, to figure out what the name of the method is you want to replace, just take the argument from the *Sensory Descriptions* table above and add "Desc" to it.

Once we've defined *ClassA4WayPath* everything else is standard, we create the room like any other, using the new class. Notice you can still use the standard *SetDesc()* method to define any sensory descriptions you haven't overridden.

The *NoExit* property

There's a shortcut available for creating maps (covered in the next section) that allows you to put a single complaint common to several directions in a compact, easy to use form. We call it the *NoExit* property. Look at the room *Brenin* to see how it works. Basically you just say:

```
Brenin.NoExit = "The valley wall bars further progress in that direction."
```

You can actually call the property anything you like, *NoExit* is short and self-explanatory, which is why we used it.

Creating A Room Map

There are two things to remember about room maps. First, all maps must be defined after *all* rooms in your game. This is because a map is nothing more than a dictionary containing rooms, and you have to define a room before you can refer to it. Python gets stuffy about things like that.

The obvious solution is to create all your rooms first, then place your map at the very end of your file. That's how we did it. At any rate, creating the map itself is child's play. Here's the map for *StartCliff*.

```
StartCliff.Map = {North:     DeepForest,
                  Northeast: DeepForest,
                  East:      A4WayPath,
                  Southeast: DeepForest2,
                  South:     DeepForest2,
                  Up: """
                      The cliff offers no handholds and to your experienced
                      eye appears totally unclimbable.
                      """,
                  Down: "It appears we're fresh out of holes to climb down.",
                  Cliff: """
                         The cliff offers no handholds and to your experienced
                         eye appears totally unclimbable.
                         """}
```

Simple, right? The name of the room, a period, and the word "Map" followed by an equals sign. Then between curly braces (not the same thing as curly brace expressions, by the way) you put one or more *entries* separated by commas. An entry is a direction (North, South, East, etc) followed by a colon, then followed by either a room reference or a string. The room will obviously be the room that direction takes you, for instance North takes you to *DeepForest*, East takes you to *A4WayPath*, and so on.

If you use a string, it will be printed when a player moves in that direction. For instance, if the player tries to go Down the game will say "It appears we're fresh out of holes to climb down.". As you can see, you may use single, double, or triple quoted text as needed.

There's something you should pay careful attention to. We included Cliff as a direction in the map! Thus if the player types "Climb cliff" the game can react appropriately. However the player still can't say "cliff" and expect something to happen, since "cliff" is an object, not a verb.

Rooms That Are Worth Points

To give a player points for discovering a room for the first time, just use the *Value* property. For instance, the *TopEastLedge* is worth 2 points when the player reaches it, thus: TopEastLedge.Value = 2

Chapter 18
How To Create Scenery

First of all, don't confuse scenery with rooms. Scenery is actually any object that the player can't take. As such scenery follows the rules for things rather than rooms. There's still some stuff you'll find familiar.

Let's take a look at the *Acorn* object you'll find at Brenin. Here are the first few lines, the line numbers aren't part of the code, only for reference:

```
1. Acorn = ClassScenery("acorn,acorns","large,huge,enormous,big")
2. Acorn.AdjectivePhrase = "huge"
3. Acorn.NamePhrase = "acorn"
4. Acorn.StartingLocation = Brenin
5. Acorn.Article = "an"
```

A few points to notice:

1. The only difference between defining a room and defining scenery (besides using a different class) is that a room definition doesn't require arguments after the class. Scenery objects (and *things* in general) require two strings separated by a comma to be inside the parentheses as you can see. The first string is a comma separated list of nouns the player can refer to this scenery object with. In this case there are two nouns a player can use: "acorn", or "acorns". Notice the fact we're using both singular and plural forms of the noun? This is a great way to get two birds with one stone. The player can say either "get acorn" or "get acorns" (meaning this scenery object) and the parser will know which object the player means. The second argument is a list of adjectives that distinguish this object from all other objects that have the same nouns. For example, if we were defining two keys, we might call one object the "gold" key and the other object the "silver" key. Note you can use the same adjectives for two different objects, as long as the nouns are different. Thus you could have a silver key and a silver rock, but not two silver keys (unless one was a big silver key and the other was a small silver key, etc).
2. The *AdjectivePhrase* property lets you pick which adjective will be used when the game describes the object. In this case we've chosen huge so the game will always call the acorns "huge" in descriptions. If you don't specify an *AdjectivePhrase* for a non-room PAWS will automatically choose the first adjective listed. This makes life easier since 99% of the time you won't have to specify an *AdjectivePhrase* if you don't want to.
3. The *NamePhrase* property for objects is a little different than for rooms. With objects it gives the object a short name. The game will always print the object with either just the *NamePhrase*, or a combination of *AdjectivePhrase* + *NamePhrase*. So the game might say "The acorn is useless" or "The huge acorn is useless". If you don't explicitly set the *NamePhrase* property PAWS will do it for you, using the first noun in the list.
4. The *StartingLocation* property lets you indicate where the object is initially located. Obviously this property is superfluous to floating objects, but for fixed scenery it's essential. If the player restarts the game all objects are moved back to the *StartingLocation*.
5. The *Article* property lets the game know which article ("a" or "an") is appropriate for this object. "A" is assumed unless you specifically change it.

Scenery Descriptions

Like rooms, scenery objects use the familiar *SetDesc()* method to set descriptions—with two additions. First, "Feel" is allowed as a description, and second, "Take" is also allowed. The "Take" description will be printed if the player attempts to take the scenery object.

You need to be careful with "Take" descriptions. Obviously, if the player types something stupid like "Take cliff" then a snide remark is only to be expected. Most players understand they aren't going to be able to take something like a cliff.

But an *acorn* is a different story. Theoretically a player can take as many acorns as they like. But the acorns don't do anything for the story, they're just atmosphere, like most other scenery objects. So we have to come up with some reasonable explanation for refusing to take it.

You'll need to be quite ingenious for some scenery!

Overriding Methods

Just like for rooms you'll occasionally want to override methods for an object. Doing so requires you create a class. See the Blockhouse scenery object for an example. Note the *ReadDesc()* method for the Blockhouse, it returns the *BlockhouseNote.ReadDesc()* method. This is an example of how one object can call another object's methods. This is very handy when you don't want to copy a complex method from one object to one or more others.

There's another benefit in this approach you might not see immediately. If I wanted to change what the Blockhouse note said I'd only have to change it in one place, not two! If there were 5 objects instead of just two you can easily see the advantage.

Chapter 19
How To Create Things

Things are defined as any object the player can pick up and carry with them. This could be something small, say a piece of jewelry, like a ring, or a weapon, or a large rock, or *anything* the player can actually carry.

Things are created just like scenery. In fact, things are just like scenery except they can be picked up. You define them with a different class, and you add a few more properties, but basically (from your point of view) they're pretty much alike. Here's the crystal mandala from *Thief's Quest* to show you what we mean. As always, the numbers are there only to match the line up to its description, they aren't part of the program.

```
1. Mandala = ClassItem("mandala","crystal")
2. Mandala.AdjectivePhrase = "crystal"
3. Mandala.Bulk = 1
4. Mandala.StartingLocation = None
5. Mandala.NamePhrase = "mandala"
6. Mandala.Value = 60
7. Mandala.Weight = 1
```

A few of the above lines are optional. Let's take a look at each line in detail:

1. You use *ClassItem* to define a thing. Like any other class you are free to make a more specialized class from this one, but most of the time you won't need to. Notice that like scenery we have two arguments, the first a comma separated list of nouns you can use to refer to the object, in this case the list contains only the word "mandala". The second argument is a comma separated list of adjectives to describe the mandala, potentially separating it from any other mandalas we may have defined. In other words, let's say you defined two keys with the noun "key". The adjective for one might be "silver", the adjective for the other "gold". Note the first argument is *required*, you must have at least one noun describe a thing so the player can refer to it. The second (adjective list) argument is optional, if you leave it out then you can only have one object with the list of nouns you supply. For example if we didn't add the adjective "crystal", there can only be one object that uses the noun "mandala".
2. This line is optional. It sets the adjective phrase the computer will use when talking about the object. "The crystal mandala is very beautiful" for example. If you don't include this line PAWS will automatically default *AdjectivePhrase* to the first adjective in the adjective list, or set it to blank if you don't include the second argument.
3. *Bulk* is the object's volume. Bulk is generally measured in cubic feet, thus the typical player's bulk is 24. (6 feet high by 2 feet wide by 2 feet deep). If you don't want to use bulk in your game just leave this line out. Bulk defaults to 0. Note that a bulk of 1 is as low as you can go and still have any bulk at all. Since the mandala is 3 inches across and an inch thick, coupled with the fact that it's made of slick crystal with no handle justifies a bulk of 1.
4. *StartingLocation* is the object you want to put this item in. It is usually a room, but might be some other type of object (like a box). Or (as we've done here) you can set the *StartingLocation* to *None*, which means it isn't "anywhere", and the player won't accidentally stumble on it.
5. The *NamePhrase* is optional, this is how the game talks about the object. If you don't include this line *NamePhrase* will automatically be set to the first noun in the noun list argument, in this case "mandala".
6. *Value* is the number of points this item will add to the score when returned to the appropriate "hoard" object. (The well house in Advent, the trophy case in Zork, etc). If not included this property defaults to 0.
7. *Weight* is how much the object weighs in "gold pieces". A gold piece is 1/10'th of a pound, so if an object weighs 10 gold pieces it weighs one pound. This is optional, if not included it defaults to 0. If you don't use weight in your game as a restriction on how much a player can carry, then just ignore this property.

In addition to these physical properties, things support the same descriptions as scenery does, using the standard *SetDesc()* method. All five sense descriptions can be supported, reasonable defaults exist if you don't set your own descriptions.

Vocabulary Limitation

Because of the way the parser deals with vocabulary, the same word can not be both a noun and an adjective. You're allowed to use the same word as a *verb* and a noun however.

For example, consider the blockhouse and the note on the blockhouse. You might consider using the word "blockhouse" as the noun for the blockhouse, and "blockhouse" as the adjective for the note ("read blockhouse note").

However, doing so will cause a fatal error when you try to reference the "noun" object. That's because the parser looks for adjectives first. It finds "blockhouse", interprets it as an adjective and suddenly has no noun where a noun was expected. This puts the disambiguation process in a tailspin and crashes your game.

On the other hand you can use the name word as a *verb* and a noun with no difficulty. Consider "north". It's a verb, and the noun of a direction object. Or "up", a verb and a preposition (as in "climb up" and "up").

Chapter 20
How To Create Actors And Monsters

An *actor* is anything the player can interact with as though it possessed some amount of intelligence. This can include anything from a small animal like a squirrel, to a sentient individual (humans or other sentients), to an object that reacts to the player's presence like a robot or automated defense systems.

A *monster* is generally any actor with combat abilities, note it need not be hostile towards the player.

Finally, the player's character itself is a monster (having combat abilities).

Actors are, quite frankly, the most difficult aspect of any game. That's why some games (like *Myst*) don't have any actors at all, others (like *Advent*) only have monsters. A good actor takes a long time to make.

Thief's Quest takes a middle ground toward actors. It has lots of monsters just for the player to fight or outwit, but it also has a basic kind of actor we call a *cameo* actor. Cameo actors make a brief appearance, utter a few lines then disappear from the game forever.

Cameo Actors

There are two cameo actors in *Thief's Quest*, the dryad and the druid. Both these actors appear on the surface. They allow only for minimal interaction with the player, the druid will strike the player if the player tries to take him, the dryad will vanish without giving her clue to the player if he tries to take her.

The only other thing they can do is respond if the player says hello. The dryad gives a very cryptic speech absolutely loaded with clues, the druid will give the player the mandala. After that, both vanish, never to be seen again.

Although if the player is observant he may catch intimations the dryad is actually still haunting the forest.

Absent Minded Player Characters

The memory system is normally completely automatic. As soon as the player sees an object, the player's character remembers that object. There are 2 cases, however, where this is not true.

1. If the player *starts* with an object in their inventory then they won't automatically remember it.
2. If the player is given an object that was previously hidden they won't remember it either.

In these cases you have to make sure the player's character manually remembers the object. Fortunately this is simple. In Thief's Quest for instance, the druid gives the player the mandala. Because the mandala is never described by the room the automatic memory system is bypassed. To manually memorize the mandala, do this:

```
Global.CurrentActor.Memorize(Mandala)
```

Actors Have Lots Of Class

Creating an actor, *without exception* first means you have to create a class to supply description and other methods. The dryad is a good example of this. In addition to the standard methods, creating a female has different responses than the default male actor that also have to be taken into account.

[... Add more Actor Material...]

Chapter 21
How To Create (and Override) Verbs

Universe comes with a wide range of verbs already defined for you, but obviously no matter how many verbs we define it will never be enough—or you may wish to change the behavior of existing verbs, perhaps adding more synonyms or whatever.

There are 3 basic situations dealing with verbs:

1. You may want to add verb synonyms to existing Universe verbs. Next to creating new verbs this is probably the most common situation. It's certainly the easiest.
2. You may want to extend or change the functionality of existing Universe verbs.
3. You want to create new verbs.

Adding Synonyms For Verbs

Let's assume for the moment that you wanted to add the synonym "xa" to the *ExamineVerb* object. There are a few possibilities.

Create New Verb Instance

First (and easiest) you could do this:

```
Examine2Verb = ClassLookAtVerb("xa")
```

This creates a new instance from an existing verb class. There's nothing wrong with this approach, it works just fine. However, it offends some programmers sense of elegance, and it does create the overhead of another instance. Not a bad solution, but not the optimum solution, either.

This *is* the solution you want to use if you're adding a new preposition that is a synonym for an existing verb. For example, if you look at the *ClassLookAtVerb()* in Universe you'll see two instances created with it, these are:

```
LookAtVerb = ClassLookAtVerb("look,l","at")
ExamineVerb = ClassLookAtVerb("examine,inspect,x")
```

As you can see, between the two instances we have 5 different synonyms for "Look at rock".

1. Look at rock
2. l at rock
3. examine rock
4. inspect rock
5. x rock

The first instance uses "at" as a preposition. The second instance doesn't use prepositions at all. Let's say you wanted to add "Look upon" as a synonym. In your game it would be perfectly acceptable to say:

```
LookUponVerb = ClassLookAtVerb("look,l","upon")
```

You've now created a new synonym, the difference is you used a different preposition. This is the only way to add synonyms with different prepositions.

Modify Universe Directly (NOT!)

NO!!!!!!!

Never modify Universe directly! This is a Very Bad Thing™. Likewise never modify PAWS directly either! The problem with this solution (apparently the simplest) is that your players may have many different games written with PAWS/Universe. If every game author starts modifying PAWS and/or Universe to make their games run better then incompatibilities are bound to occur. And that means headaches for the players, and thus they won't want to play your games.

It's very difficult to make new versions of PAWS/Universe that don't break games written in older versions, but it is possible—if you wrote the code originally. ☺

If you have to have a change that you feel will benefit PAWS or Universe and make it better for everyone then email me (wolf@one.net) with your change. Ideally you should have a file containing your change(s), along with a detailed description of what the change does and how it might impact the rest of the system (assuming you have suspicions it might).

I'm always looking to improve the system, and welcome input from readers. If your idea is used I'll give you credit in the source code. Since everyone has a copy of the source code anyone can review it and suggest changes. This is how most open source projects work, there's a single person (or small group) that coordinate changes to the core system, and anyone that wants to can submit changes.

Replacing Existing Objects

This approach will work with any object created by Universe, but will usually be restricted to verbs, which make up the vast majority of Universe objects.

In our example, here's the "correct" solution:

```
DeleteObjectFromVocabulary(ExamineVerb)
ExamineVerb = ClassLookAtVerb("examine,inspect,x,xe")
```

The *DeleteObjectFromVocabulary()* function scans all the vocabulary related dictionaries and removes *ExamineVerb* from them. Once this is done and the second line executed there's no trace of the original verb object, it's replaced with the new one, which differs only by the addition of a new synonym.

Creating A New Verb

Creating a new verb isn't difficult, but you have to create a new class to do it. Let's examine the *ListenTo* verb to see how a verb is created. We stripped the comments to make the code shorter, and the preceding numbers and roman numerals are, of course, not part of the code, they're simply there for reference..

```
1. class ClassListenToVerb(ClassBasicVerb):
   2. """Verb to handle Listen To Rock"""

   3. def SetMyProperties(self):
         a. ClassBasicVerb.SetMyProperties(self)
         b. self.ObjectAllowance = ALLOW_MULTIPLE_DOBJS + ALLOW_NO_IOBJS
         c. self.OkInDark = TRUE

   4. def Action(self):
         a. """Listen To action"""
         b. if len(Global.CurrentDObjList) == 0: return Complain("Listen to
            what?")
         c. for Object in Global.CurrentDObjList:
              i.  Object.MakeCurrent()
             ii.  Object.MarkPronoun()
            iii.  Object.DescribeSelf("SOUND")
         d. return TURN_ENDS

5. ListenToVerb = ClassListenToVerb("listen","to")
```

This should look pretty familiar to you. A verb is nothing more than a class. You have to define 2 specific methods, *SetMyProperties()* (the same method you set for any other class in PAWS) and *Action()*, which is unique to verbs.

Notice that *ClassListenToVerb* is descended from *ClassBasicVerb*. All verbs you create will normally come from *ClassBasicVerb*, unless you specifically want to descend from another verb (which you normally wouldn't unless extending a verb, see below).

Defining The Class

Lines 1 and 2 define the class, these work just like any other class definition. Verbs generally won't have services and will be descended from *ClassBasicVerb*. Line 2 is the documentation string that Python's documentation program uses to create automatic documentation.

Defining *SetMyProperties()*

Line 3 defines the *SetMyProperties()* method. Line 3a calls the *SetMyProperties()* method of the ancestor *ClassBasicVerb*, just as with every other class you've ever created in PAWS.

Line 3b is the first real line. It sets the *ObjectAllowance* property, which is just adding together of the objects allowed for this verb. ObjectAllowance controls how many direct and/or indirect objects the verb can handle. For example the *QuitVerb* doesn't allow any direct or indirect objects to be used with it. The *ListenToVerb* allows multiple direct objects but no indirect objects. For example, you can't say *Listen to rock with stethoscope*.

The only rule is that if you don't allow any direct objects you can't allow any indirect objects either. This makes sense when you think about it because the parser (or a human!) would have no way to tell which objects were direct objects and which were indirect without a preposition.

To define which objects a verb is allowed you add constants together. The following lines are all legal:

```
ALLOW_NO_DOBJS + ALLOW_NO_IOBJS
ALLOW_ONE_DOBJ + ALLOW_NO_IOBJS
ALLOW_ONE_DOBJ + ALLOW_ONE_IOBJ
ALLOW_ONE_DOBJ + ALLOW_MULTIPLE_IOBJS
ALLOW_MULTIPLE_DOBJS + ALLOW_NO_IOBJS
ALLOW_MULTIPLE_DOBJS + ALLOW_ONE_IOBJS
ALLOW_MULTIPLE_DOBJS + ALLOW_MULTIPLE_IOBJS
```

However the following two lines are not:

```
ALLOW_NO_DOBJS + ALLOW_ONE_IOBJS
ALLOW_NO_DOBJS + ALLOW_MULTIPLE_IOBJS
```

Line 3c means that this verb can be used in the dark. You don't need light to listen to something, but you do need light to travel by. *ClassBasicVerb* normally assumes that a verb needs light, line 3c changes that assumption.

Defining The *Action()* method

This is the meat of the verb, the line that actually *does* whatever it is the verb does. *Action()* is called by parser assuming all the boring details where done correctly (the player typed a valid command). Line 4 and 4a define the method.

Line 4b checks to see if there are any direct objects. Direct objects that have been parsed are held in *Global.CurrentDObjList*. So if the player said *Listen to rock and roll* the first direct object would be *Rock*, and the second would be *Roll*. (This always assumes there are objects in your game called rock and roll, of course ☺).

Returning TURN_ENDS or TURN_CONTINUES

If there are no direct objects we return with a complaint "Listen to what?". This brings up an important point about the *Action()* method. It must return either TURN_ENDS or TURN_CONTINUES. Return TURN_ENDS if you want *Engine.AfterTurnHandler()* to run, TURN_CONTINUES if you don't. The *Complain()* function always returns TURN_CONTINUES, to make coding complaints simpler.

Returning TURN_CONTINUE means the action took so little time that a full turn shouldn't pass. Daemons won't run, fuses won't get shorter, and so on.

Returning TURN_ENDS, on the other hand, means that a considerable amount of time has passed (1 turn) and that daemons should run and fuses burn down. If you examine the various verbs in the Universe library you'll see this pattern. Verbs that let the player travel from place to place return TURN_ENDS, while system verbs like *SaveVerb* and *RestoreVerb* return TURN_CONTINUES.

You'll notice *ListenToVerb* returns TURN_ENDS, indicating that listening takes a turn. This is consistent with most adventure games, that assume if you're listening it takes a while. You can always extend this verb to return TURN_CONTINUES, if you like. We show you how in the next section.

Processing Each Object

If your verb deals with a single direct object that object will always be stored in *Global.CurrentDObjList[0]*. If (as in this case) the verb is capable of dealing with multiple direct objects then you need to loop through them with a FOR Object IN loop.

This neat little trick means that each time through the loop *Object* will refer to the current direct object from *Global.CurrentDObjList*. If the player referred to 5 direct objects then the FOR loop will repeat 5 times, and *Object* will refer to each direct object in turn.

Line 4c(i) is required to make the *Self()* function (used in curly brace expressions, or CBE's) work correctly when verbs deal with multiple direct objects. Line 4c(ii) sets the him/her/it pronouns correctly for the object in question when multiple direct objects are used.

Of course line 4c(iii) is the pay off. It tells the object to describe the sound the object is making.

Returning SUCCESS or FAILURE

Line 4d returns TURN_ENDS, which indicates the *Action()* method took 1 turn to execute. Returning TURN_CONTINUES here indicates that the *Action()* method was instantaneous, 1 turn hasn't passed yet so the end of turn processing shouldn't happen. Remember, the end of turn processing happens only when your *Action()* method (or the methods it calls) return TURN_ENDS.

Creating A Verb Instance

Once the class is defined you can create as many instances of the verb as you need, generally one per unique combination of verb and preposition. The *LookAtVerb* instance and the *ExamineVerb* instance are both created with *ClassLookAt*, but one uses the *at* preposition and the other uses no preposition at all.

Verb Creation Summary

Does this seem like a lot of work to create a verb? It really isn't. The verb is an action, not a piece of narrative, and thus has to be created with programming. The Universe library contains many verbs already created for you, so you may not have to create your own verbs at all.

But should you have to, remember that a verb *does* something, not just describes something. That means a new set of instructions for the computer, and that means programming. Of course if you look at the verbs in *Universe* you'll probably be able to copy one as a template to get you started.

Extending Existing Verb Functionality

This is actually a combination of creating a new verb and deleting an existing vocabulary object.

Let's use our *ClassListenToVerb* as an example. You already know this verb returns TURN_ENDS when used, which means each time the player listens to something the daemons will run and the fuses will burn down. You may not want listening to take much time, so you want to have the *ListenToVerb* object return TURN_CONTINUES instead of TURN_ENDS. Will you have to create a whole new verb?

No, you can simply extend Universe's existing *ClassListenToVerb* and make it return TURN_CONTINUES. Here's the code:

```
1. class ClassTQListenToVerb(ClassListenToVerb):
2. """Verb to handle Listen To Rock"""

3. def SetMyProperties(self):
       a. ClassListenToVerb.SetMyProperties(self)

4. def Action(self):
       a. """Listen To action returns FAILURE"""
       b. ClassListenToVerb.Action()
       c. return TURN_CONTINUES

5. DeleteObjectFromVocabulary(ListenToVerb)
6. ListenToVerb = ClassTQListenToVerb("listen","to")
```

Ok, let's look at this. Lines 1 and 2 define *ClassTQListenToVerb*, which is the name of our extended class. Notice it's descended from *ClassListenToVerb* instead of *ClassBasicVerb*. This means we can inherit all properties of *ClassListenToVerb* without explicitly resetting them.

Lines 3 and 3a define *SetMyProperties()*, which only has one line, calling *ClassListenToVerb's SetMyProperties()* method (this is the same thing every class you'll ever create does).

Lines 4 and 4a define the *Action()* method. Line 4b is the trick. We are calling the *Action()* method defined in *ClassListenToVerb*, which you've already seen—but we're ignoring the return value, the parser never sees the TURN_ENDS normally returned.

Line 4c returns TURN_CONTINUES, which tells the parser *not* to run the end of turn routines.

Line 5 deletes the original *ListenToVerb* object from PAWS's vocabulary. This is required because we're in effect replacing the *ListenToVerb* object with the one created in line 6.

Chapter 22
How To Create Containers

Unlike other game systems *any* class in PAWS can be used to create containers. A container is just an object that can contain other objects. In Thief's Quest, for example, the flashlight is a container.

There are two approaches to making a container. The first is to use the two specific container classes, *ClassShelf* (to make flat surfaces where other objects can (visibly) sit) and *ClassContainer* to make a normal container like a jar or a box.

The other approach is to create a new class basing it on *ClassScenery* or *ClassItem* and one of the 4 container services, *ServiceContainIn*, *ServiceContainOn*, *ServiceContainUnder*, or *ServiceContainBehind*.

Simple Containers

Here's how to make a simple shelf and a simple cardboard box using the *ClassShelf* and *ClassContainer*. They won't be very interesting versions (you need to add more descriptions for that) but they are fully functional. The shelf will hold 3,200 pounds and 500 cubic feet of objects, and objects sitting on the shelf will be visible when the shelf is mentioned, and the box can hold 500 cubic feet and 3,200 pounds of objects that will *not* be visible unless the box is looked into.

```
MagicShelf = ClassShelf("shelf")
MagicShelf.MaxBulk = 500
MagicShelf.StartingLocation = StartCliff

Box = ClassContainer("box","brown,cardboard,card,board")
Box.MaxBulk = 500
Box.StartingLocation = StartCliff
```

Scenery As Container

Here's an example of how the hook is implemented in Roger Firth's demo game *Cloak Of Darkness*, courtesy of Neil Cerutti. Notice the hook is *scenery*, it's described as part of the long description of the room. The long description given here is for the command *examine hook*.

```
#---------------
# The Brass Hook
#---------------

class ClassHook(ServiceContainOn, ClassScenery):
    def SetMyProperties(self):
        ClassScenery.SetMyProperties(self)
        ServiceContainOn.SetMyProperties(self)

    def Enter(self, Object):
        """ Turn on the light in the Foyer Bar when the player puts the cloak
on the hook. """
        if Object == Cloak: Bar.SetIsLit(TRUE)
        return ClassScenery.Enter(self, Object)

Hook = ClassHook("hook,peg","small,brass")
Hook.NamePhrase = "hook"
Hook.AdjectivePhrase = "small brass"
Hook.MaxBulk = 1
Hook.MaxWeight = 10
Hook.StartingLocation = Cloakroom

Hook.SetDesc("Take", "The hook is screwed to the wall.")

Hook.SetDesc("L", """
                  It's just a small brass hook,
                    (choose(clock in self).contents, " with a clock hanging on
it.","  screwed to the wall."))
                    """)
```

Chapter 23
Daemons, Fuses And Recurring Fuses

The title of this chapter may have totally confused you. Fuses? Daemons? What's that all about?

Relax. Programmers have extremely quirky senses of humor. A *daemon* is nothing more than a function that runs at the end of every turn. A *fuse* is a function that runs after a given number of turns have elapsed. And a *recurring fuse* is just a function that runs every *X* turns.

For example, PAWS has a function called *GameDaemon()*. This function is run at the end of every turn, all it does is increment the number of turns. This points out the fact that a function running at the end of every turn need not be complex to be useful. In fact, here's the code for *GameDaemon*.

```
def GameDaemon():
    """Daemon to handle standard game chores"""
    Global.CurrentTurn = Global.CurrentTurn + 1
```

As you can see, the only thing the function does is add 1 to *Global.CurrentTurn*! Also notice it has no arguments. By definition no function used as a daemon, fuse, or recurring fuse can have arguments. It must be completely self contained.

So what's the difference between a daemon, a fuse, and a recurring fuse? The answer is: *none*. At least, not in the function being run. The *GameDaemon* function above can be run as a daemon, a fuse, or a recurring fuse without any changes to the function itself.

The difference is in how the function is *started.*

Starting A Daemon

For example, you start a daemon with this line of code:

```
StartDaemon(GameDaemon)
```

Or you can also say:

```
StartDaemon(GameDaemon,0)
```

The second argument is the number of turns before (or between) running the function. In a daemon's case it is run every turn (0 turns delay before running and 0 turns between each run).

Starting A Fuse

You start a fuse with this line of code:

```
StartDaemon(GameDaemon,5)
```

The second argument is the number of turns to delay before running the function, in this case 5. Fuses only run once, and then stop automatically.

Starting A Recurring Fuse

A recurring fuse is run with this line of code:

```
StartDaemon(GameDaemon,-5)
```

The second argument is the number of turns between each of the function's runs. Notice the second argument is a negative number. That distinguishes a recurring fuse from a normal one. In fact, you may have already guessed what's coming—a daemon, fuse, and recurring fuse use the *same mechanism* to run. The *only* difference in starting them is the second argument. If missing or 0 it's a daemon, if positive it's a fuse, and if negative it's a recurring fuse.

Stopping A Daemon, Fuse, Or Recurring Fuse

If you want to stop a daemon, fuse, or recurring fuse prematurely use this line of code:

```
StopDaemon(GameDaemon)
```

Obviously, once a fuse is run it's stopped automatically (fuses only run once after a given number of turns). You can easily stop recurring fuses or daemons that have already run with this code, however.

Resetting A Daemon, Fuse, Or Recurring Fuse

You can also *change* the delay on a fuse that hasn't run yet, or recurring fuses whether they've run or not. In fact, you can change a daemon to a fuse or recurring fuse or vice-versa. The mechanism used is very flexible. Just repeat the *StartDaemon()* line. For example, to change the *GameDaemon()* function from running every turn to running every other turn just say:

```
StartDaemon(GameDaemon,-1)
```

In other words, wait 1 turn between runs of *GameDaemon()*. This would allow the player to move twice as quickly, doing two actions in one turn instead of one action per turn.

What's the Point?

Why do you need daemons, fuses, or recurring fuses? You've already seen one usage, a turn counter. In Thief's quest all the monsters act on the end of the turn, controlled by a daemon. Fuses are handy for one time events the player triggers. One example might be a *literal* fuse, the player accidentally starts the timer on a bomb! Recurring fuses are helpful for repetitive or cyclical events, for example a room door that only opens every 3 turns for instance.

Once you start thinking of how a delayed, recurring, or repetitive event can be handy to have the more uses you'll come up with. PAWS daemon system is extremely simple to use, making daemons and fuses an attractive option.

Chapter 24
Debugging Your Game

Let's face it, nobody's perfect. When you create your game you're going to make mistakes. And finding those mistakes can be particularly frustrating when you have no idea why a problem is happening.

Professional programmers use an array of sophisticated tools and techniques to help them find and eliminate errors. These range from graphical debuggers to literally going to another programmer and saying "help!" ☺

But you aren't a professional programmer. You don't have the money to buy expensive tools, especially just to write games that you'll never make any money from. Likewise you likely won't have the luxury of going to another PAWS programmer and asking for help. So what do you do?

PAWS Debug Mode

The first weapon against bugs in your arsenal is a special mode in PAWS called the *debug mode*. To use it all you have to do is make sure your set up game function (remember *TQUserSetup()*?) has the line:

```
Global.Production = FALSE
```

This enables the debug mode. Once your game is finished and debugged you change the line above to TRUE instead of FALSE. This disables the debug mode and prevents knowledgeable players from cheating.

To actually use the debug mode you have to turn it on. Just type "debug" when your game is running and PAWS will respond *Debug verb is active. Debug is ON*.

The Parser Trace

From then on every time you type a command you'll get a parser trace. A parser trace lets you see how PAWS is parsing the command you typed. For example, here's the parser trace from the command "get rock" at the beginning of *Thief's Quest*. Note the player's typing is in bold, everything else is the computer's response.

```
At 'Start'

You are standing in a small clearing in a pine forest. Behind you (to the
west) is a cliff stretching upward at least a thousand feet. About a third of
the way up you can plainly see the word "Start" carved in letters fifty feet
high. The cliff is slightly hollowed out so that it seems to menace you like a
crashing wave of granite. To the north and south you can see trackless pine
forest. To the east a faint trail leads away from the cliff and deeper into
the forest.
There is a small grey rock here.
> debug
 Debug verb is active Debug is ON

> get rock
 rock is Noun
Testing Allowed Objects
    small grey rock passed
Testing Known Objects
    small grey rock passed
Testing Visible Objects
    small grey rock passed
Testing Reachable Objects
    small grey rock passed
Testing Favored Objects
    small grey rock failed
    small grey rock failed unambiguously
The rock emits a bright green flash when you pick it up, but does nothing else
remarkable.

>
```

Let's take a look at the trace and see what it's showing you. First, it says that "rock" is a noun. Then it tells you it's testing "allowed" objects. These are objects the verb (in this case VerbTake) has on its list of allowed objects. Next the trace says that *small gray rock passed*. This means the rock was on the allowed item list for VerbTake. Or (more usually) that VerbTake didn't have a list of specific items it worked with. In Universe none of the verbs have objects on their allowed lists.

Next the parser does the same thing for *known objects*, objects the player has already seen and knows about. Technically objects that are known are in the player's Memory[] list. As you can see the small gray rock passed this test as well.

The next two tests are for *visible objects* and *reachable objects* respectively. As you can see the parser tries a series of tests to eliminate ambiguous objects. Normally ambiguous objects are eliminated silently, without telling the player anything. However, if the *final* object on a list is eliminated (meaning none are left) then the parser will complain to the player with the appropriate error message.

The last test is a little harder to explain, as you can see the rock failed this test. That's ok! This test is used when two identical objects (normally invisible) are in the room at the same time. If no object has *ParserFavors* set to TRUE, all objects are allowed. This is best illustrated by examining the cliff in debug mode:

```
> examine wall
 wall is Noun
Testing Allowed Objects
    non-existent wall passed
    wall passed
    granite cliff passed
Testing Known Objects
    non-existent wall passed
    wall passed
    granite cliff passed
Testing Visible Objects
    non-existent wall passed
    wall failed
    granite cliff passed
Testing Reachable Objects
    non-existent wall passed
    granite cliff passed
Testing Favored Objects
    non-existent wall failed
    granite cliff passed
It's your typical 2,000 foot unclimbable (granite) cliff.
```

First, there are 3 objects in the game named "wall", the regular wall object, the non-existent wall (that appears in rooms without walls), and the cliff (valley wall). Ok, all 3 walls are allowed, and known. However, the regular wall object (wall) isn't visible and is eliminated. The other two are reachable, but only the cliff is favored. Had the cliff not been favored, both it and the non-existent wall would have failed the favored test, but because *both* the objects failed the favored test both would have been used, and both object descriptions would have printed. This would have said something like:

```
There's no wall here.
It's your typical 2,000 foot unclimbable (granite) cliff.
```

This is clearly undesirable! That's why the *Cliff* object sets the *ParserFavors* property to TRUE.

You normally won't have to worry about the parser, or the *ParserFavors* property, PAWS and Universe normally handle all this between them. But it's sometimes handy to see at what stage of the parsing process objects are eliminated, this can point out errors in your game logic.

Saying CBE's

Now that the debug mode is active you can use the verb *say* to actually print out CBE's (Curly Brace Expressions) that you type in. For example let's say we wanted to see the location of the dryad. Here's what we'd do:

```
> say {Dryad.Where()}
Testing Allowed Objects
Testing Known Objects
Testing Visible Objects
Testing Reachable Objects
<TO.ClassOutside instance at 00879B1C>
```

```
>
```

Oops. PAWS gave us the location as expected, all right. Except it did so in *Python*, not in *English*. Even I have no idea which particular room this is, and I wrote the game! Here's what you should say instead:

```
> say (Dryad.Where().NamePhrase)
 Testing Allowed Objects
Testing Known Objects
Testing Visible Objects
Testing Reachable Objects
Maple Stand
>
```

Much better! When trying to identify objects always make sure you're using the object's NamePhrase property. This will let you identify which objects you're dealing with in terms you'll recognize.

Saying Object *Lists* with CBE's

If you wanted to find out what the player knows about then you could say:

```
> say (Global.Player.Memory)
 Testing Allowed Objects
Testing Known Objects
Testing Visible Objects
Testing Reachable Objects
[<TQ.ClassTree instance at 0089735C>, <TQ.ClassTree instance at 00898004>,
<TQ.ClassTree instance at 0089820C>, <TQ.ClassTree instance at 008983C4>,
<TQ.ClassTree instance at 0089848C>, <TQ.ClassTree instance at 0089840C>,
<TQ.ClassTree instance at 008985CC>, <TQ.ClassTree instance at 0089870C>,
<TQ.ClassTree instance at 008988C4>, <TQ.ClassTree instance at 008988EC>,
<TQ.ClassTree instance at 0089E32C>, <Universe.ClassItem instance at
0088F014>, <TQ.ClassCliff instance at 00894F94>, <TQ.ClassTQPlayer instance at
0088E904>]
>
```

Oops again. PAWS gave us the contents of the player's memory in Python, not English. Not good. So we have to use one of the purpose-built debugging functions, designed especially to display lists of objects, called *DebugPassedObjList()*. Like so:

```
> say (DebugPassedObjList("Memory",Global.Player.Memory))
 Testing Allowed Objects
Testing Known Objects
Testing Visible Objects
Testing Reachable Objects
Memory
-->stream
-->beech leaf
-->beech tree
-->birch leaf
-->birch tree
-->maple leaf
-->maple tree
-->pine cone
-->pine needle
-->pine resin
-->pine tree
-->small grey rock
-->granite Cliff
-->me
None
>
```

Ah hah! The first argument of the function is just text you want to appear at the top of the list. The second argument is the object list, in this case the player's memory. As you can see a number of objects are already known at the beginning of the game. Most of these have been defined with the *ClassLandmark* class, which automatically puts objects created with it into the player's memory.

This brings up a point. The only objects that will automatically be placed in the player's memory are those listed individually when a room is first entered. The small gray rock, for example.

Items which are scenery will have to be added to a player's memory manually. In the example of *ClassLandmark* the following line of code was added to the *SetMyProperties* method.

```
Global.Player.Memorize(self)
```

Note this only works for objects that the player is likely to know about already, such as trees. For objects that the player won't know about until he sees them, (ones defined as scenery) you'll just have to create a new class that adds the room's *Contents* list to the player's memory when the room is entered.

Setting Variables

To set a variable in the debug mode you type:

```
Say (DoIt("Global.Debug=FALSE"))
```

In other words the *DoIt()* function takes whatever is inside quotes and tries to execute it as though it were a line of Python code. This is a very powerful capability, of course it's also very dangerous, so use it with caution!

Turning Debug Mode Off

To turn off debug mode just type "debug" and PAWS will respond *Debug verb is active. Debug is OFF*. In other words, the debug command is a toggle. Use it once to turn debug on, use it again to turn debug mode off.

Other Debugging Options

You do of course have a few more options when faced with intractable bugs and elusive solutions. First, you can learn to use Python's IDLE tool. IDLE is a graphical debugging tool to develop Python programs. Teaching IDLE is beyond the scope of this text, and quite frankly I've never had to use it. In other words, I don't know how. ☺

Another option is to post your questions on the rec.arts.int-fiction newsgroup. The folk there are among the friendliest and most helpful on the net, always willing to help a new author overcome the odd conundrum. Sometimes, just writing the problem down will give you insight into the solution because explaining it to someone else sometimes clarifies the problem in your own mind.

Finally, if you believe the problem may lie with PAWS itself and not your game you can drop me a line at wolf@one.net. Because PAWS is a continual work in progress it's very possible bugs may have slipped through the beta-testing process. Don't be afraid to report what you believe to be bugs. The more bugs that get fixed in PAWS, the better it will be for all game authors using the system, right?

I'm also always on the look out for suggestions about new features for the next version of PAWS!