Python Adventure Writing System Technical Manual Version 1.0

Written By Roger Plowman

Table Of Contents

Introduction

Who Should Read This Book

Unless you're a programmer who wants to change PAWS itself, you won't need this book. You should read *Writing Interactive Fiction in PAWS* instead. That book teaches you how to write a game in PAWS and doesn't get too technical so is ideal if you want to just write games in PAWS.

An Overview

PAWS is short for *P*ython *A*dventure *W*riting *S*ystem. Like its sister languages TADS, Inform, Hugo, Alan, and ADVSYS (among others) PAWS's sole purpose in life is to create text adventures. All PAWS games (like PAWS itself) are written in the Python programming language.

Python was created (and is still maintained) by Guido van Rossum. It is a simple, elegant, and powerful language that was originally intended to write UNIX shell scripts.

Python proved so popular, however, that it soon spread to other venues, and is now a popular stand-alone programming language that has been ported to a variety of operating systems, including:

- MS-DOS
- Windows 3.x, 9x, NT, CE, 2000
- OS/2
- Macintosh
- Amiga
- Psion
- BeOS
- Linux
- UNIX (many flavors)
- QNX
- VMS (DEC VAX computers)
- VxWorks (embedded systems, like VCRs!)
- Palm OS
- Acorn RISC-OS.

As a result, PAWS and any games written with PAWS won't have to be ported to these systems, you can simply copy the compiled games and they'll run! (Assuming the target system has Python 1.5.2 or later). The most recent compatible version of Python is version 2.0. Versions 1.5.2 and 1.6 will also run PAWS games without problem.

PAWS Components

PAWS consists of 7 primary components:

- This manual
- A tutorial manual
- The game engine
- The Universe library
- The "play" module
- A large, complex sample game called Thief's Quest (which is currently incomplete).
- A small simple sample game called *Cloak Of Darkness*. This game is useful because it's been written for many different IF languages. Thanks to Roger Firth for creating the original game, and Neil Cerutti for creating the PAWS version.

Game Engine

The game engine is fairly powerful, but lacks many features considered essential in today's languages. Not to worry, the Universe library (or a replacement for Universe) supplies these.

The engine is in the PAWS.py file. It contains:

- Important game constants (such as TRUE and FALSE)
- A parser capable of translating English into objects using very basic disambiguation
- A global variables object (mainly for the parser)

- Useful functions (such as intersect and union)
- a base verb object
- a base "thing" object

By itself the engine isn't very useful. It's designed to be used with a library (Universe, by default) to provide much of the higher level functionality one would expect in an adventure writing system.

Because it's designed to work seamlessly with a library, almost any engine method can be easily "swapped out" by the library. You can even replace the default parser with your own version! This might be useful when writing games in languages other than English, for example.

Universe Library

The Universe library is similar to the Inform library, ADV.T or the WorldClass library. It basically defines the game "world", providing such basic services as the ability for objects and rooms to describe themselves, verbs like "get", "save", "quit", and so forth.

Universe is designed to create a few basic object types, which can be vastly enhanced by a selection of mix and match services. For example, scenery is just a basic thing with a fixed item service. However, you can sniff, touch, listen to and look at the scenery—without programming at all!

The "play" module

To allow easy running from the Python command line we include the "play" module. You never have to change it, just include it with the rest of the PAWS modules. This provides all the functions that have to appear in the top level, such as save and restore.

Thief's Quest Game

Unlike many text adventure development systems, PAWS comes with an enormous, extremely well commented, sample game. *Thief's Quest* is also a lot of fun to play. (Even if I do say so myself!) It's contained in TQ.py. To run the game from the command line type "Python play TQ". Note the capitalization!

Chapter 1
The Game Engine

The PAWS game engine consists of a handful of objects, each with their own properties and methods. This chapter will introduce you to them.

Constants

Constants are values that are known in advance and never change. The following constants have been defined in PAWS:

ALLOW_MULTIPLE_DOBJS	Used when setting the verb's ObjectAllowance. Allows the verb to have multiple direct objects.
ALLOW_MULTIPLE_IOBJS	Used when setting the verb's ObjectAllowance. Allows the verb to have multiple indirect objects.
ALLOW_NO_DOBJS	Used when setting the verb's ObjectAllowance. Forbids the verb from having *any* direct objects. (For example, the "quit" verb has no direct objects). When this constant is used you must always use the ALLOW_NO_IOBJS as well.
ALLOW_NO_IOBJS	Used when setting the verb's ObjectAllowance. Forbids the verb from having *any* indirect objects. (For example, the "look at" verb has no indirect objects.
ALLOW_ONE_DOBJ	Used when setting the verb's ObjectAllowance. Allows the verb to have one and only one direct object.
ALLOW_ONE_IOBJ	Used when setting the verb's ObjectAllowance. Allows the verb to have one and only one indirect object.
ALLOW_OPTIONAL_DOBJS	Used when setting the verb's ObjectAllowance. Allows the verb to have none, one, or more direct objects.
BINARY_PICKLE	Saved games will be stored in a binary format, unreadable by humans.
DAEMON	Used internally by the daemon/fuse system to identify a function that's a daemon. It's unlikely you'll ever need to use this constant directly.
FAILURE	Returned by functions that failed for some reason. A synonym for FALSE.
FALSE	**The opposite of true. Often used in testing for true/false conditions. A synonym for FAILURE.**
FINISHED	Used to set the Global.GameState. It means the game is finished, and tells the game to shutdown and return to the operating system. When the player types "quit" (or "exit", or whatever) the Global.GameState changes to FINISHED.
FUSE	Used internally by the daemon/fuse system to identify a function that's a fuse. It's unlikely you'll ever need to use this constant directly.
HER	Defines the numeric key into the PronounsLisaDict table for "her".
HIM	Defines the numeric key into the PronounsLisaDict table for "him".
IT	Defines the numeric key into the PronounsLisaDict table for "it".
RECURRING_FUSE	Used internally by the daemon/fuse system to identify a function that's a recurring fuse. A recurring fuse is a fuse that automatically rearms itself once executed. It's unlikely you'll ever need to use this constant directly.
RUNNING	Used to set the Global.GameState. It means the game is currently running.
STARTING	**Used to set the Global.GameState. It means the game has not yet completed its start up phase. As soon as the game completes the start up phase Global.GameState changes to RUNNING.**
SHALLOW	Used with the *ContentDesc()* method to indicate a description should be *shallow*, in other words non-recursive. A deep contents description lists an item's contents if the item is open or transparent, a shallow description does not, even if for example a box were open. This makes it easier to say "There's a box, a lamp, and a chair here." Without mentioning that the (open) box has contents.

SUCCESS	Returned by functions that succeed. A synonym for TRUE.
TEXT_PICKLE	Saved games will be stored in text format, which is (somewhat) readable by humans.
THEM	Defines the numeric key into the PronounsLisaDict table for "them".
TRUE	The opposite of FALSE. Often used in testing for true/false conditions. A synonym for SUCCESS.
TURN_CONTINUES	Returned by verb *Action()* method when you do *not* want the end of turn handling (such as daemons and fuses) to occur yet. Actions that take little time (such as taking an object or saving a game) should return this value. TURN_CONTINUES is a synonym for FAILURE.
TURN_ENDS	Returned by verb *Action()* method when you *do* want the end of turn handling (such as daemons and fuses) to occur. Actions that take significant time (movement, examining scenery and objects, etc) should return this value. TURN_ENDS is a synonym for SUCCESS.

Utility Functions

The PAWS.py file contains functions that are independent of the engine itself. If you follow our convention of using "from PAWS import *" in your library, you can use the functions exactly like built-ins (that is, you don't have to put "PAWS." in front of them.

The functions, arranged alphabetically, are:

AppendDictList(*Dict, Key, Value*)	This function is intended to be used by dictionaries that use *lists* as their values. The first time a key is added to the dictionary the value (a single value) is placed in the dictionary as a single item list. Each time this function appends a value thereafter to a key, the value is appended to the existing *list* of values. This function is intended primarily to add nouns, adjectives, verbs, and prepositions to the parser dictionaries, although there's nothing stopping you from using it if you need to.
Choose(*Decision, TrueChoice, FalseChoice*)	*Decision* must evaluate to zero (FALSE) or non-zero (TRUE). If *Decision* is true then the value in *TrueChoice* is returned, otherwise the value in *FalseChoice* is returned. This function is intended mainly to be used in CBE's inside strings, meaning *TrueChoice* and *FalseChoice* will be strings.
ClearScreen()	Clears the screen by the simple expedient of printing enough lines to scroll any existing text completely off the screen.
Complain(*Text*)	Prints a string of text to the screen, appends a line break, and returns TURN_CONTINUES. By returning a *Complain* function you can easily print some text to the screen and return a failure code at the same time. Very handy, and used a *lot* throughout the system. Remember, since TURN_CONTINUES is a synonym for FALSE you can return *Complain()* whenever you want a FALSE value as well.
DebugTrace(*Text*)	Prints the string in *Text* but only if *Global.Debug* is TRUE.
DebugDObjList()	Prints the names of the objects in *Global.CurrentDObjList*, but only if *Global.Debug* is TRUE.
DebugIObjList()	Prints the names of the objects in *Global.CurrentIObjList*, but only if Global.Debug is TRUE.
DebugPassedObjList(*Message, List*)	This function prints the string in *Message*, then prints the *Object.\$Desc()* for each object in *List*. Note that *List* must be a list of objects. This function does nothing unless
DeleteDictList(*Dictionary, Object*)	Deletes Object from the passed dictionary list. This function is the opposite of *AppendDictList()*.
DeleteObjectFromVocabulary(*Object*)	This function deletes an object from the vocabulary. You use it if you plan to override an object from the world library (Universe, for instance, with your own object. Generally you do this to extend verb vocabulary, but you can also use it with any object created by the world library.
DisambiguateList(*List, TestMethod, ErrorMethod, [Actor]*)	Takes an object list and tries to figure out which object in the list is actually the one the player means when they type in a object's name but don't include an adjective. This function is called by DisambiguateListOfLists(), it's never called directly. DisambiguateListOfLists() passes a list of ambiguous objects (all objects in the list have the same noun), the test method to winnow out objects, and the error method to print out the error method if *all* objects on the list are eliminated. • *List* – List of ambiguous objects to disambiguate. Will *always* be a list since this function isn't called for a single object. (Single objects are never ambiguous). • *TestMethod* – Method returning a true or false value when applied to a single object within List. Examples include

	IsReachable() and *IsVisible()*. • *ErrorMethod* – Method returning a string error message if no ambiguous objects return TRUE for the test method. String is printed to the screen by *DisambiguateList*. • *Actor [Optional parameter]* – Included only if the test method requires *two* objects (such as *IsVisible()*) to determine TRUE or FALSE. (An object has to be visible *to* another object, that other object is the actor).
DisambiguateListOfLists(*ListOfLists*, *TestMethod*, *ErrorMethod*, *[Actor]*)	Takes an object list of lists (like the parser produces for Global.CurrentObjLists) and tries to figure out which objects the player means when they type in a object's name but don't include an adjective. • *ListOfLists* – List of ambiguous objects to disambiguate. This list can include a combination of single objects (unambiguous) and lists of objects (ambiguous). • *TestMethod* – Method returning a TRUE or FALSE value when applied to a single object. Examples include *IsReachable()* and *IsVisible()*. • *ErrorMethod* – Method returning a string error message if no ambiguous objects return TRUE for the test method. String is printed to the screen. • *Actor [Optional parameter]* – Included only if the test method requires *two* objects (such as *IsVisible()*) to determine TRUE or FALSE. (An object has to be visible *to* another object, that other object is the actor).
DoIt(*CodeString*)	This function allows you to execute Python code contained in the string *CodeString*. In other words, *DoIt("Global.Debug = FALSE")* would turn off the debugging features built into the game engine. This function is helpful during a debugging session, because you can use it to execute assignment statements inside {}'s with the "Say" verb. (See the chapter on debugging in *Writing Interactive Fiction In PAWS*).
GameDaemon()	This function is actually a daemon, a function that is run by the end of turn handler. All it does at present is increment the turn counter *Global.CurrentTurn*.
GetPlayerInput()	This function is part of the basic parsing system. It prompts the player to type in a command, then breaks that line into one or more commands and places them in Global.CommandsList. You probably won't need this function unless you plan to rewrite the parser.
HandlePunctuation(*WordList)*	This function is also part of the parsing system. It takes a list of words (generally a command) and looks for words that end with punctuation marks. If there are any, it splits the punctuation mark from the end of the word and makes it a new item in the list. This function can be useful if you're getting input directly from the player, it lets you take the word list and make punctuation marks their own "words" in the list.
Indent(*Level*)	Returns *Level* * 3 spaces. This function is useful when creating a string of recursively nested items that need to be placed in an indented outline format (such as an inventory).
Intersect(*List1*,*List2*)	Returns a list of items common to both *List1* and *List2*. This can be thought of as a bit-wise *and* that applies to lists.
InVocabulary(*Word*)	Returns TRUE if the string *Word* is in one of PAWS vocabulary dictionaries (nouns, verbs, prepositions, etc.), FALSE if it isn't.
ParserIdentifyNoun(*Start*, *End*)	This function is the heart of the parsing system. It takes the words from Global.ActiveCommandList from the *Start* word to the *End* word and returns a list of objects that match the noun and any

	adjectives used. This function does *no* disambiguation, if there are 3 stones in the game and the player types "Get rock" then this function might (for instance) return [Boulder, SmallRock, BlueRock], even if the player is nowhere near the boulder and can't see the small rock! You'll probably never call this function direcly.
RunDaemons()	This function runs all daemons when called, it shortens fuses and runs them when the fuse expires, and rearms recurring fuses once run. This function is part of the *default_AfterTurnHandler()* function.
Say(*Text*)	This is a replacement for the print statement. It takes a single string argument which is the text you want to print to the screen. It properly handles word wrapping at the edge of the screen and if *Text* is longer than the screen will pause with a "[--More--]" message to allow the player to read everything. It supports \n as a line break and \m to force a "more" message. In addition, it supports placing *any* Python expression that can be rendered into text between curly braces ("{" and "}"). This allows you extreme amounts of flexibility when creating your text.
SCase(*Sentence*)	Returns a string with the first character in upper case and the remainder of the string in lower case. This is called *sentence case*. Since all strings in PAWS are stored internally in lower case, this function provides an easy way to print an entire sentence properly capitalized.
Self()	Returns a reference to the current object, used in CBE's (see *Say()* above).
SetRemove(*List1*, *List2*)	Returns a list of items where *List1* has had all items in *List2* removed.
StartDaemon(*DaemonFuse*, *FuseLength*)	This function adds a daemon, fuse, or recurring fuse to the Global.DaemonDict dictionary. In other words you use it when you want to start up a daemon or arm a fuse. A daemon/fuse is just a function that PAWS runs automatically, either every turn (a daemon) or once after a specified delay in turns (a fuse) or repeatedly, pausing a given number of turns between each run (a recurring fuse). Let's assume our function is called *ActorScript()*. Let's further assume we're first going to set it up as a daemon (to run every turn), as a fuse (to run once after a 5 turn delay), or as a recurring fuse (to run every 5'th turn). Here's the call to start *ActorScript()* as a daemon: *StartDaemon(ActorScript)*. Note you don't put the parentheses after the name of the daemon! This passes the function's address. To run *ActorScript()* as a fuse with a 5 turn delay do this: *StartDaemon(ActorScript,5)*. In other words, the only difference between making a daemon and a fuse is putting a number after the function argument. This will delay 5 turns, run *ActorScript()* once, and remove it from the list of active daemons/fuses. To run *ActorScript()* every 5 turns do this: *StartDaemon(ActorScript, -5)*. In other words, use a *negative* number for the delay. This function returns SUCCESS unless *DaemonFuse* isn't a function (you can't schedule object methods, only functions). If *DaemonFuse* isn't a function, *StartDaemon()* returns FAILURE.
StopDaemon(*DaemonFuse*)	This function removes a daemon or recurring fuse from the list of active daemons. It will also remove a fuse that's still burning (one that hasn't run yet). This function returns SUCCESS if *DaemonFuse* is found on the

	active daemon list (*Global.DaemonDict*) and FAILURE if *DaemonFace* isn't on the list, or isn't a function.
Union(*List1*, *List2*)	Returns a list of items containing all of *List1* and elements of *List2* that aren't in *List1*. This can be thought of as a bit-wise *or* that applies to lists.

Default Game Handlers

The default game handlers are the "logic cycle" of the game. Although they are arranged alphabetically in the table below, they are really called in the following order:

```
Engine.GameSkeleton()          # Called by play module
Engine.SetUpGame()             # Called by Engine.GameSkeleton

While Game Running             # Repeat these steps until game is done
   Engine.PreTurnHandler()     # Just before player types command
   Engine.Parser()             # Let player type command & parse it
   Engine.TurnHandler()        # Do whatever player said
   Engine.AfterTurnHandler()   # Do this when turn ends, else repeat loop

Engine.PostGameWrapUp()        # Game over commentary to player
sys.exit()                     # Shut down game & Python, return to OS
```

Note the use of indirect function calls, the functions in the table below are assigned to properties inside Engine. The names of the PAWS supplied handlers always begin with the prefix *default_* so it's easy to see which functions go with which Engine properties.

default_AfterTurnHandler()	This function does nothing but run daemons and fuses by calling the *RunDaemons()* function. You shouldn't have to replace it in most games.
default_GameSkeleton()	This function handles the entire game logic cycle listed above. You shouldn't have to replace it for your games.
default_Parser()	This function is the PAWS parser. It handles all aspects of getting typed input from the player and translating it into objects the game can use. Unless you plan to create a game in a language that places adjectives after the noun (such as Spanish) you shouldn't have to touch the parser at all.
default_PostGameWrapUp	This function does nothing, you'll definitely want to replace it for each game you write. This function is intended to give the player a post game wrap up of how they did during the game. Did they die? What was their score? That sort of thing.
default_Preparse()	There are certain things that have to happen to the input text before the parser can be allowed to get at it. This function takes the exact text the player typed (preserving case) and places it in *Global.SaidText*, which is used by the debugging system in Say() commands. It then forces the command to lower text and scans it for words not in the game's vocabulary. If any are found it complains. Finally it translates pronouns into the appropriate words. You shouldn't have to replace this function unless you also replace the parser.
default_PreTurnHandler()	This function does nothing, you'll want to replace it for each game you write. The pre-turn handler is intended for actions that happen very quickly (such as combat, poison that acts over seconds rather than minutes, etc). Basically if you want a player's commands to count against them *regardless of whether the command was valid or not*, then you put the code here instead of in a daemon. See Thief's Quest for an example.
default_Prompt(*PromptArg*)	This function serves two purposes. First, it resets the screen row to row + 1 and it resets the screen column to 1. Then it returns the appropriate string to act as the prompt for the player, by default "> ". You probably won't have to replace this function unless you want to take advantage of the function's argument for some more elaborate prompting system.
default_SetUpGame()	This function does nothing by default, it is replaced by Universe's *UniverseSetUpGame()* function.
default_TurnHandler()	This function does nothing except call *Global.CurrentVerb.Execute()*,

	you shouldn't have to replace this function. Now the parser calls this function only after it has successfully parsed the player's command. Calling it yourself (or rather calling *Engine.TurnHandler()*) is probably not a good idea.
default_UserSetUpGame()	This function does nothing by default. You replace it in every game you write since this function (or rather *Engine.UserSetUpGame()*) is specifically intended to give you a place to customize game setup.

ClassFundamental (Abstract class, not instantiated)

This class is the "root" of all classes in PAWS. It provides a couple of methods that are used by all classes.

Get(*Attribute*)	This method returns either *None* if the attribute doesn't exist, or the contents of a property or the return value of a method that has no arguments. For example *Rock.Get("NamePhrase")* would return the same thing as *Rock.NamePhrase*, while *Rock.Get("LDesc")* would return the same thing as *Rock.LDesc()*. This function is intended to blur the line between properties and methods that have no arguments, a la TADS. It's also very useful to call an object when you aren't sure if the object will have the attribute or not. If you say *Rock.Color* and *Rock* doesn't have a *Color* attribute, your game will crash. On the other hand, *Rock.Get("Color")* simply returns *None* when *Rock* doesn't have the *Color* attribute.
MakeCurrent()	This method sets *Global.CurrentObject* to self. For instance, *Rock.MakeCurrent()* would set *Global.CurrentObject* to *Rock*. This method is called whenever an object is the direct object of a command, or when it's being described. You probably won't ever use this method directly, it's mainly used for the *Self()* function..
SetMyProperties()	This method actually does nothing in this class, it's simply here as a placeholder, to guarantee that all objects will have this method. It's used in descendants to set "default" properties for instances. That means that when objects are instantiated they'll already have values for any properties set in this method.

ClassParserError (Instantiated as ParserError)

This object stores all the messages the parser will print as properties. In alphabetical order they are:

DobjsNotAllowed	*"This verb can't have any direct objects."* Printed when a verb has been defined not to have any direct objects, but the player typed direct objects. "Quit" is an example of such a verb.
IobjsNotAllowed	*"This verb can't have any indirect objects."* Printed when a verb has been defined not to have any indirect objects, but the player typed indirect objects. "Look at chest" is an example of a verb that doesn't allow indirect objects.
MultipleActors	*"You can only tell one thing at a time to do something."* Only one actor is allowed per command. This error would be printed if the player typed something like "Fred, Barney, drive to Bedrock".
MultipleVerbPrepositions	*"PROGRAMMING ERROR: Two or more verbs share this verb and preposition combination."* This error is printed when the parser detects two or more verbs that have the same combination of verb and preposition, for instance if two different verbs were defined as "look at" or "quit".
NoPreposition	*"That verb needs a preposition."* Printed when the player uses a verb that requires a preposition, but didn't supply one. For example, "Dig trench" would generate this error because "dig" requires "with".
NoPreviousCommand	*"You haven't done anything yet!"* Printed when the player types "again" as the very first command of the game.
NotInVocabulary	*"I don't know the word %s."* Printed when the player types a word that isn't in the game's vocabulary.
NoSuchVerbPreposition	*"I don't recognize that verb/preposition(s) combination"* The player typed a valid verb, but used a preposition with it that isn't supposed to be part of the verb, for example: "Quit To DOS" would generate this error, since "quit" doesn't have a preposition.
NoVerb	*"There's no verb in that sentence."* Printed when the parser can't find any word that it recognizes as a verb.

ClassGlobal (Instantiated as Global)

This object contains all the global information, information required by every part of the program. The PAWS engine uses the Global object for parser information almost exclusively. Alphabetically the global properties are:

ActiveCommandList	Holds the text of the command currently being parsed. You probably won't need this list unless you're planning to write a *PreParse()* method for the Engine.
AdjsDict	Dictionary of adjectives associated with objects. Appended each time a "thing" object is created.
Again	The "again" verb object. Set to *None*, it is the responsibility of the Universe (or other game) library to set this property to the proper verb object. (In Universe it's *AgainVerb*)
ArticlesList	List of all English articles (a, an, etc).
CommandBreaksList	List of all syntax elements that mark the end of a command. (Used to break up multiple commands on a single line.)
CommandsList	A list of all commands the player typed in on a single line, one command per list element. Once parsing is complete, this list will be empty. (When element 0 is copied to the *Global.ActiveCommandList* it is deleted from this list).
ConjunctionsList	A list of all English conjunctions (and, a comma, etc).
CurrentActor	The object that the parser has determined the player was commanding to do something. If no actor was commanded, the player's object will be the current actor.
CurrentDObjList	Holds a list of direct objects successfully parsed from the current command. Only the most basic disambiguation has been performed by the engine. Additional disambiguation will be performed by the Universe library and/or the author's game.
CurrentObject	Contains the object under consideration by the verb action.
CurrentIObjList	Holds a list of indirect objects successfully parsed from the current command. Only the most basic disambiguation has been performed by the engine. Additional disambiguation will be performed by the Universe library and/or the author's game.
CurrentPrepList	Holds a list of all prepositions successfully parsed from the current command. This is a list of *strings*, not objects.
CurrentPreviousVerb	The verb executed by the "again" command.
CurrentScreenColumn	The screen column the cursor is currently on. Defaults to 1, this value is used by the *Say()* function to know when to word wrap.
CurrentScreenLine	The screen line the cursor is currently on. Defaults to 1. This value is used by the *Say()* function to know when to print a "more" message and pause.
CurrentVerb	The object the parser determined is the verb of the current command. This is an *object*, not a string.
CurrentVerbNoun	The exact word the player typed for the verb. For example, if the QuitVerb's command can be either "quit" or "exit", and the player typed "quit", this variable holds the string "quit".
DaemonDict	This dictionary holds all active daemons, fuses, and recurring fuses. Each entry has a function address as a key, with a list of 2 values as the entry. The first number is the number of remaining turns before the daemon, fuse, or recurring fuse is activated. For daemons this value is always 0, for fuses and recurring fuses this value will be positive. This value can never be negative. The second item in the list is the *original* fuse length passed to the *StartDaemon()* function. This number can be positive, 0, or negative. If positive then the entry is a fuse, if 0, a daemon, and if negative a recurring fuse.
Debug	Setting this value to TRUE allows you to use the *Debug()*, *DebugD()bjList()*, and *DebugI()bjList()* to print debugging displays that won't appear if Debug is

	FALSE.
DisjunctionsList	List of all English disjunctions ("but", "except", etc).
GameState	The current status of the game, can be STARTING, RUNNING, or FINISHED.
MaxScreenColumns	The maximum number of columns on the screen. Defaults to 80.
MaxScreenLines	The maximum number of lines on the screen. Defaults to 25.
NounsDict	Dictionary of all nouns associated with "things". Appended each time a "thing" object is created. (Verb objects use the VerbsDict dictionary, however).
Player	The object that represents "me" in the game. The object the player "is", while playing. The Engine sets this value to None, it is the responsibility of the Universe library (or game library) to set this value to an object.
PrepsDict	Dictionary of all prepositions associated with verb objects. Appended to each time a verb object is created.
Production	TRUE when the game is released to the general public, FALSE while the game is under development. When TRUE the debugging system is disabled, and the *SayVerb* object will translate all {'s and }'s to |'s and |'s respectively. This keeps players from using the debug system to cheat.
PronounsDict	Dictionary of all pronouns (he, she, it, them, etc) and the objects currently associated with them. Appended each time a verb object is created.
PronounsListDict	List of all English pronouns and the key values (used in the PronounDict) associated with them.
SaidText	The literal text that was after the verb. This is used by ClassSayVerb.
VerbsDict	Dictionary of all verbs used in the game. Appended each time a verb object is created.

ClassEngine (instantiated as Engine)

This object is the heart of PAWS, the actual game engine. It contains all the methods and properties necessary to run the game. Methods are listed with arguments but properties need none.

AfterTurnHandler	Property to hold the address of the actual AfterTurnHandler method.
GameSkeleton	Property to hold the address of the actual GameSkeleton method.
PostGameWrapUp	Property to hold the address of the actual PostGameWrapUp method.
PreParse	Property to hold the address of the actual PreParse method.
PreTurnHandler	Property to hold the address of the actual PreTurnHandler method.
Prompt	Property to hold the address of the actual Prompt method.
RestoreFunction	Property to hold the address of the Restore function (set in game skeleton). Defaults to None. The actual restore function is defined in the play module.
SaveFunction	Property to hold the address of the Save function (set in game skeleton). Defaults to None. The actual save function is defined in the play module.
SetUpGame	Property to hold the address of the actual SetUpGame method.
UserSetUpGame	Property to hold the address of the actual UserSetUpGame method.
TurnHandler	Property to hold the address of the actual TurnHandler method.
Version	The version of the *Engine*. This can be different from the version of the Universe library, and will certainly be different from the version of the game.
XlateCBEFunction	Property to hold the address of the actual TranslateCBEFunction, which is defined in the play module.

ClassBaseObject

This object is the base class for all "thing" objects (objects the player can touch or take or examine). Its purpose is to provide a mechanism to populate the NounsDict and AdjsDict dictionaries for the parser. In addition it provides the IsKnownToPlayer property, which the parser uses for basic disambiguation.

ClassBaseObject requires two arguments, one a comma delimited string of nouns that apply to the object, and the second (optional) argument a comma delimited string of adjectives. For instance:

```
SmallRock = ClassBaseObject("rock,stone","small,grey,gray")
```

Any class or instance derived from this class will require the same two arguments.

ClassBaseVerbObject

This object is the base class for all verb objects the game author will create. It's more sophisticated than ClassBaseObject, but, then again, it does more. Like *ClassBaseObject* this class automatically appends the verb and preposition to the appropriate dictionaries.

Like *ClassBaseObject* this class also requires two comma delimited strings, one of verbs, the other of prepositions. For example:

```
LookAtVerbs = ClassBaseVerb("look","at")
QuitVerbs = ClassBaseVerb("quit,exit")
```

Action()	This method "does" the verb's action. For example, the Quit verb might set the Global.GameState. Action returns *SUCCESS* if you want the AfterTurnHandler() to run, FAILURE if you don't. The version found in PAWS.py is simply a place holder. Every verb will override this method.
Execute()	Called by the parser, this method performs a layered disambiguate (generic disambiguate from the Engine, and a library supplied specific disambiguate, either of which can abort the command by returning FAILURE instead of *SUCCESS*). Only when both disambiguates succeed will it launch the Action method. Note it's almost never necessary to override this method in descendents.
GenericDisambiguate()	This method performs a very generic disambiguate. It prevents specific disambiguation if there are no direct/indirect objects. It removes objects from the Global.CurrentDObjList/Global.CurrentIObjList that the player doesn't know about yet (IsKnownToPlayer is FALSE) or (if the OnlyAllowedxObjLists are defined) removes all objects not on the allowed lists. Finally, it verifies that verbs that don't allow direct/indirect objects actually lack them.
ObjectAllowance	This property determines if none, one, or multiple direct or indirect objects are allowed by the verb. For example, "quit" allows no direct or indirect objects, "look at" allows no indirect objects, "get" allows multiple direct objects but no indirect objects, and so forth.
OKInDark	This property is FALSE by default, TRUE if verb can be performed in the dark.
OnlyAllowedDObjList	If this list contains any objects, then only objects on this list are allowed as direct objects to the verb. Others will be removed.
OnlyAllowedIObjList	If this list contains any objects, then only objects on this list are allowed as indirect objects to the verb. Others will be removed.
SpecificDisambiguate()	This routine is always replaced by the library, it may also be replaced by specific verbs. This method performs the final disambiguation, usually by checking to see if the object is present, visible, and reachable. (Details may vary for different libraries or specific verbs).

The Universe Library

This section of the manual is the largest. The Universe library is extensive, we're going to split up the library into one chapter per functional grouping. These groupings are:

- Universe constants and utility functions
- ParserError addenda
- Global object addenda
- Individual classes/instances (one chapter for each)

Chapter 2
Universe Constants And Functions

The constants and functions in this chapter are available to the game author (you) for use in your game's library. (Remember, the game you write will be in a single .py file, called the *game library*). References to the *library* usually mean Universe, *game library* means the game file you write.

Constants

UniverseCopyright	Our copyright, don't change it, remove it, or prevent it from printing out in your game. Universe is free, but we want people to know it's been used in your game. This is the "price" for using Universe.
UniverseVersion	The current version of the Universe library. This can be handy when determining which version of the library your games work with. We strive to keep Universe versions backward-compatible, but won't guarantee it! See above. ☺

Functions

The functions in Universe fall into two categories—replacement methods for the Engine, and freestanding functions. If you see an argument called *self* you know it's a replacement for an Engine method.

Agree(*Verb, Subject, Contract*)

Given a string verb, a subject object, and TRUE/FALSE for contraction, this function returns the correct form of the verb as a string. *Subject* defaults to the current actor, and *Contract* defaults to FALSE. So Agree("is") returns "are", since "you" is the assumed subject and it isn't a contraction. See also Be(), Have(), Do() and Go().

Be()

Be() is a short form of *Agree*("be", Global.CurrentActor, FALSE).

Do()

Do() is a short form of *Agree*("do", Global.CurrentActor, FALSE).

Go()

Go() is a short form of *Agree*("go", Global.CurrentActor, FALSE).

Have()

Have() is a short form of *Agree*("have", Global.CurrentActor, FALSE).

IncrementScore(*Amount, Silent*)

This function increments the player's score, and if *Amount* is non-zero, tells the player. If *Silent* is TRUE then it doesn't inform the player *regardless* of the value of *Amount*.

Me()

This function is the short form of Global.CurrentActor.FormalMe.

UniverseBanner()

This function prints the Universe copyright banner. Please don't change it.

Universe_SetUpGame()

This is Universe's replacement method for the Engine's default_SetUpGame(). It performs the following actions:

- Move all objects to their starting locations and calculate the maximum score for objects and rooms.
- Populate the Global lists *ActorList*, *FloatingLocationList*, and *LightSourceList* with objects that are those particular types.
- Calls the *Engine.UserSetUpGame()* method. This method is written by the game author as part of their game.
- Calls the *Game.PrintGameIntroduction()* method to print the banners, game introduction text, and generally get the player ready to play.
- Adds the nouns "all" and "everything" to the dictionary to handle commands like "get all".
- Start the Game Daemon.
- Memorize *Ground*, *Sky*, *Wall*, and *NoWall* objects. (There's no current actor at the time these objects are defined, so the normal method doesn't work).

You()

The short form of Global.CurrentActor.FormatYou.

Your()

The short form of Global.CurrentActor.FormatYour.

Youm()

The short form of Global.CurrentActor.FormatYoum.

Chapter 3
Additional Parser Errors

The following errors have been added to the *ParserError* instance to allow specific disambiguation to print the appropriate messages. Where the characters *%s* appear in the text below, *%s* will be replaced by the appropriate phrase. An example phrase is always provided, generally for the "small gray rock" object.

Nonesense	***"That doesn't make sense"*** Printed when the player tries to use an object with a verb that isn't on the verb's allowed object lists.
NotADirection	***"You can't go that way."*** Used as a last resort by the Travel() method to indicate the player can't go that direction.
NotAnActor	***"You have lost your mind."*** This message is printed when the player tells a non-actor to do something. For example, "Rock, get the wand".
ObjectNotHere	***"There's no %s here."*** This message is printed when the player tries to manipulate an object that isn't present. Example: "There's no rock here."
OnlyOneDObj	***"You can only use one direct object with this verb."*** The player tried to use more than one direct object with a verb that only allows one direct object.
OnlyOneIObj	***"You can only use one indirect object with this verb."*** The player tried to use more than one indirect object with a verb that only allows one indirect object.
TooDark	***"It's too dark to see how."*** Printed when a player tries to do something in the dark and the verb used can't be done in the dark.

Chapter 4
Additional Global Properties

To allow the game author easy access to Universe data, several new properties have been added to the *Global* object instance. In alphabetic order they are:

ActorList[]	A list of all actor objects. Handy when you need to scan through all actors.
AllObjectsList[]	A list of all "thing" objects (all objects descended from *ClassBaseObject*).
CompassList[]	A list of the 8 direction objects (North, Northeast, etc).
CurrentScore	The player's current score.
CurrentTurn	The player's current turn. A turn typically lasts 5-10 minutes, you should decide exactly how long you want a turn to be in your game. (Always assuming the passage of time is particularly important to your game).
DefaultMap	This is the map used when no other map is available. It contains an entry for every direction object defined by Universe, and the string "You can't go that way." Note this dictionary is the "court of last resort", in most well-designed games this map is unnecessary, it's only for quick and dirty games, or to keep the game from crashing because a map location isn't available due to a bug.
FloatingLocationList[]	All objects that "float". In other words, all objects whose location is equal to the player's location. This is useful for things like the sky, the ground, floors, ceilings and walls, or objects you want the player to have the definitions for, i.e. "What is a griffon?"
LightSourceList[]	A list of all objects that can potentially produce light. Objects on this list need only to be capable of producing light, they need not be currently lit!
LitParentList[]	List of rooms that are currently being illuminated by lit light source objects. This list is recalculated every turn by the AfterTurnHandler().
Restarting	TRUE/FALSE property. Set to TRUE when the game is restarting, this prevents the game introduction from being repeated.
Verbose	TRUE/FALSE property. If set to TRUE then the room's long description will be printed every time the room is entered, if FALSE (the default) then the room's long description is printed only the first time the room is entered or the player types the command "look".
VerbAgreementDict	This dictionary holds exceptions to the general rules concerning subject/verb agreement. Each key is the form of the verb being checked (be, have, etc). If the verb is prefixed with the word "contracted" (contractedbe, contractedhave) then it will be used by the *Agree()* function when that function has a TRUE *Contract* argument. Each entry in the dictionary is a list of 2 strings, the first for plural usage, the second for singular usage. If a verb isn't in this dictionary *Agree()* assumes it follows the normal rules of subject/verb agreement.

Chapter 5
ClassGameObject

This class (instantiated as *Game*) creates an object that lets you customize your game easily and quickly. The object has the following properties and methods (methods have at least one argument (*self*) in parentheses:

Author	The name of the game's author (you).
Banner(*self*)	Returns the game banner.
Copyright	This property holds the game's copyright year, for example: "1998, 1999". You don't need to include the "©", Universe does that for you.
IntroText	This property holds the game introductory text, which can be quite long.
Name	The name of your game.
PrintGameIntroduction(*self*)	This function actually prints the *IntroText* property. You may want to override this method with one of your own.
Version	The version of your game. Note this is a string, and doesn't include the word *version*. It defaults to "1.0".

Chapter 6
ClassBasicThing

This class is descended from PAWS *ClassBaseObject*, the root of the "thing" hierarchy. A "thing" is any object the player can perceive as a separate entity, a room, or a direction. Thus "sword", "rock", "forest", and "north" are all "things".

Generally (but not always) a "thing" can be touched. The exception would be directions, which are the only abstract "thing" in the Universe library. We'll talk about directions in their own chapter.

ClassBasicThing is a very complex class, there are almost 100 properties and methods associated with it!

This is because things have so many different behaviors and attributes. A short list of categories (by no means complete!):

- Sensory Descriptions (sight, smell, sound, odor, taste)
- Physical Attributes (bulk, weight, transparency, etc)
- Parts of Speech Descriptions (long description, short description, "the" description, etc)
- Mapping Attributes (location, contents, etc)
- And so forth...

In order to more easily digest this morass of properties and methods we're going break the properties and methods into separate tables.

Properties

In the following table the word "self" refers to the object under discussion. For example, if we were talking about a rock, then self is a rock. The following table contains all the class properties in alphabetical order.

AdjectivePhrase	Adjective phrase used in short description. This is the actual wording you want to use, for example "small gray". The description need not include every adjective used when you instantiated the object, nor do you place commas between the adjectives in the phrase.
Article	The article ("a" or "an" in English) used to refer to self. It defaults to "a", but you have to be careful to match the correct word. It's "an umbrella", but "a red umbrella". Or "an orange umbrella".
Bulk	The bulk of self. Bulk has no counterpart in the real world, we recommend you consider 1 unit of bulk equal to 1 cubic foot. This would make the player (on average) 24 cubic feet (6' x 2' x 2'). Bulk is a measure of how big something is.
CantLookBehind	TRUE/FALSE. Defaults to FALSE, it determines whether the player can look behind self.
CantLookInside	TRUE/FALSE. Defaults to TRUE, it determines whether the player can look inside self.
CantLookOn	TRUE/FALSE. Defaults to FALSE, it determines whether the player can look on self.
CantLookUnder	TRUE/FALSE. Defaults to FALSE, it determines whether the player can look under self.
ContainerPrepositionDynamic	Defaults to "inside". Preposition used when putting object in/under/behind/on another object. The "active" preposition.
ContainerPrepositionStatic	Defaults to "inside". Preposition used when describing how an object holds its contents.
Contents	This is a list of objects contained by self.
FormatYou	Defaults to "" (an empty string). This is the word used to say "you" for an object. It's defined here to make coding the You() function simpler. FormatYou probably won't be defined for the vast majority of "things".
HasFloatingLocation	TRUE/FALSE. False by default, set this property to TRUE if self's location is a

	method.
IsActor	TRUE/FALSE. False by default, set this property to TRUE if self is an actor. (Can be given commands).
IsBlatantOdor	TRUE/FALSE. False by default, set this property to TRUE if self has an odor so intrusive it should be mentioned each time the object is described.
IsBlatantSound	TRUE/FALSE. False by default, set this property to TRUE if self makes a sound so intrusive it should be noted each time the object is described.
IsBroken	TRUE/FALSE. False by default, set this property to TRUE if self is broken. Broken affects self's long and short description.
IsHer	TRUE/FALSE. False by default, set to TRUE if self is female. If neither IsHim nor IsHer are true the object is neuter.
IsHim	TRUE/FALSE. False by default, set to TRUE if self is male. If neither IsHim nor IsHer are true the object is neuter.
IsLightSource	TRUE/FALSE. False by default, set to TRUE if self has the potential to produce light. (Compare with *IsLit* below.)
IsLiquid	TRUE/FALSE. False by default, set to TRUE if self is a liquid.
IsLit	TRUE/FALSE. False by default, set to TRUE if self is actually producing light. Compare with *IsLightSource* above. Note: if this property can ever become TRUE you must set *IsLightSource* to TRUE.
IsOpen	TRUE/FALSE. False by default, set to TRUE if self is open. Used by the *Enter()* method to determine if one object can enter another.
IsOpenable	TRUE/FALSE. False by default, set to TRUE if self is openable. Note that an object can be open without being openable. Openable implies self can be either open or closed.
IsPoisonous	TRUE/FALSE. False by default, set to TRUE if self is poisonous. The exact meaning of poisonous differs from object to object.
IsPotable	TRUE/FALSE. False by default, set to TRUE if self is edible or drinkable.
IsScenery	TRUE/FALSE. False by default, set to TRUE if self is scenery. (Scenery is just there to add atmosphere. Scenery returns `""` when *HereDesc()* is called.
IsTransparent	TRUE/FALSE. False by default, set to TRUE if self is transparent. Important for determining if the object can pass light or be seen into when closed.
Location	Object's location. This is either None (if the object is nowhere) or another object.
MaxBulk	The maximum bulk self can contain.
MaxWeight	The maximum weight self can contain. Generally this is used by Actors to determine how much they can carry.
Memory	A list of objects this object knows about. Memory is almost always reciprocal, if an actor knows about an object, the object knows about the actor. This is helpful during disambiguation.
NamePhrase	The name of an object as told to the player. For example, "rock". Note the NamePhrase should be a single word.
ParserFavors	FALSE by default, TRUE only when two identical objects will occupy the same location at the same time. Currently this is the case only for user-defined walls, since the *NoWall* object appears when the *HasWall* property is FALSE, which you need to set when you create a custom wall. When defining a custom wall always set this property to TRUE.
StartingLocation	Self's starting position, this must either be None or another object.
Value	How much the object or room changes the player's score. The player gets an object's value for placing it in the board, or for discovering a room.
Weight	How much the object weighs. Doesn't normally apply to rooms, and never applies to direction objects.

Description Methods

A "description" method is one that describes the object with a particular part of speech. For example, one description method returns "a rock", another returns "the rock", and so forth.

It's important to note that each of these methods returns a *string*, none of them actually print anything to the screen. They are used exclusively to build sentences that will later be printed.

Further note that every description method ends with *Desc*. It's guaranteed that any method ending with *Desc* is a description method and always returns a string.

For the sake of example, our object will be a small gray rock, with the *NamePhrase* "rock", and the *AdjectivePhrase* "small gray".

Therefore Rock.ADesc() would return "a small gray rock". Further assume that the rock is in a forest, and (potentially) in a glass box. (You'll see why below).

ADesc(*self*)	Returns *"a small gray rock"*
AmnesiaDesc(*self*)	Returns *"I don't ever remember seeing a rock around here."* Used in both the DefineVerb and disambiguation.
ArticleDesc(*self*)	Returns *"a"*
CantLook(*self*)	Returns "*It's impossible to look {inside/under/on/behind} the object.*" The appropriate word is used, depending on what the player typed.
CantReachDesc(*self*)	Returns *"You can't reach the small gray rock."* If the rock is inside something transparent it returns *"You can't reach the small gray rock. You'll have to open the glass box first."*
CantSeeDesc(*self*)	Returns *"You can't see any rock here."* This function is used in disambiguation.
ChooseArticleDesc(*self*)	Returns either *"the rock"* or *"a rock"* as appropriate. If the current actor has met this object before, we use "the" otherwise "a".
ContentDesc(*self,Level, Shallow*)	This method returns one of 3 strings: If the object can't hold objects (MaxBulk is 0): "" (returns an empty string) If the object is empty: *"The glass box is empty."* Otherwise: *"The glass box contains:"* followed by a nicely indented list of the box's contents. If *Shallow* is TRUE, the contents of objects listed are *not* listed, even if the object is open or transparent. This method is slightly different. You call it like this: *GlassBox.ContentDesc()*, but the method will make further calls to nested objects, passing the nesting level as the second argument. For example, if there's a gem inside a boule inside the box the following calls are made: *GlassBox.ContentDesc()*, *BouleContentDesc(1)*, *Gem.ContentDesc(2)*.
ContentsPrefixDesc(*self*)	Returns *"The glass box contains:"* This function is called by *ContentDesc()*, it's unlikely you'll ever call it directly.
ContentsShallowDesc(*self*)	Returns *"The bag contains a bell, a book, a candle, and a glass box."* It creates a more natural look than *ContentDesc()*, but doesn't list the contents of any transparent or open objects.
DontSeeInteresting(*self*)	Returns *"You don't see anything interesting {under/behind/inside} the rock."* Of course this method uses the actual word appropriate to the player's action.
EmptyDesc(*self*)	Returns *"The glass box is empty."* This function is called by *ContentDesc()*, it's unlikely you'll ever call it directly.
FeelDesc(*self*)	Returns *"It feels like an ordinary rock to me."* Notice it doesn't include the adjective phrase.
GroundDesc(*self*)	Returns *"It looks like ordinary ground to me"* Even though this method really only applies to rooms it's placed in *BasicThing* so

	handle rooms that are also objects.
HereDesc(*self*)	Returns *"There's a small gray rock here."*
InsertedDesc(*self, Object, Spontaneous*)	Returns *"The rock goes inside the bag"* if *Spontaneous* is TRUE, otherwise *"You put the rock inside the bag."*. *self* in this case would be the bag, *Rock* would be *Object*. This method is used to describe placing one object inside/on/behind/under another. The appropriate preposition is used automatically.
LookDeepDesc(*self*)	Returns *"In the box you see a key and an envelope."* This function uses *ContentsShallowDesc()*, and is called by "look in box" and the like.
LDesc(*self*)	Returns *"It looks like an ordinary rock to me."* Notice it doesn't include the adjective phrase.
MultiSDesc(*self*)	Returns *"small gray rock:"*. This method is intended to be used when listing several objects. It's very similar to the *SDesc()* method, but appends a colon to the string.
NoDesc(*self*)	Returns "" (an empty string). There are cases where it is useful to know that you can output a string that won't change the appearance of another string. In other words, this method deliberately returns a string of 0 length so that when appended to another string the first string is unchanged.
NotWithVerbDesc(*self*)	Returns *"That doesn't make sense."* Used in disambiguation as an error method.
OdorDesc(*self*)	Returns *"It smells like an ordinary rock to me."* Notice it doesn't include the adjective phrase.
PluralDesc(*self*)	Returns *"small gray rocks"*
PronounDesc(*self*)	Returns *"it"*, *"him"*, or *"her"* depending on the settings of the *IsHim* and *IsHer* properties.
ReadDesc(*self*)	Returns *"You can't read a small gray rock."*
SDesc(*self*)	Returns *"small gray rock"*
SkyDesc(*self*)	Returns *"It looks like ordinary sky to me"* Even though this method really only applies to rooms it's placed in *BasicThing* to handle rooms that are also objects.
SoundDesc(*self*)	Returns *"The small gray rock isn't making any noise."*
TasteDesc(*self*)	Returns *"It tastes like an ordinary rock to me."* Notice it doesn't include the adjective phrase.
TheDesc(*self*)	Returns *"the small gray rock"*.
WallDesc(*self*)	Returns *"It looks like ordinary wall to me"* Even though this method really only applies to rooms it's placed in *BasicThing* to handle rooms that are also objects.
WrongPrepositionDesc(*self, Object, Spontaneous*)	Returns *"The rock can't go inside the table!"* if *Spontaneous* is TRUE, *"You can't put the rock inside the table."* if Spontaneous is FALSE. *Object* in this case is *Rock*, while *self* is *Table*.

Other Methods

We can lump all the other methods of *ClassBasicThing* together. These methods handle a variety of tasks, many of which have to do with physical properties. Many others have to do with "paths", that is, the path one object must take to reach another.

Paths are expressed in containers. For instance, assume you have a rock inside a bottle inside a box in the forest.

There are basically 4 kinds of paths, the *theoretical path*, the *physical path*, the *light path*, and the *visibility path*.

The theoretical path from the rock to the forest passes through the bottle and the box. It assumes nothing can get in the way. You can always have a theoretical path between two objects (in the same room), unless one or both objects have a location of *None* (meaning they are nowhere, they don't "exist").

The *physical* path requires that all containers between the two objects be open. So if either the box or the bottle is closed there is no physical path between the rock and the forest.

The *light path* is the path a beam of light would travel between the two objects. It requires all containers between the two objects be either open or transparent.

The *Visibility path* requires there be a light path between the two objects *and* that a source of light is present along the light path.

The sense of sight requires a visibility path (line of sight), the other 4 senses require a physical path.

A path never searches beyond one room, so objects in another room don't have a physical or light path, although they might have a theoretical path.

AllowedByVerbAsDObj(*self*)	Returns TRUE if the GlobalCurrentVerb allows this object to be used as a direct object. This is a test method for the disambiguation routines. It returns true if this object is in a verb's AllowedDObjList.
AllowedByVerbAsIObj(*self*)	Returns TRUE if the GlobalCurrentVerb allows this object to be used as an indirect object. This is a test method for the disambiguation routines. It returns true if this object is in a verb's AllowedIObjList.
CheckActor(*self*)	Returns TRUE if self is an actor, prints an error message and returns FALSE if not.
ContentBulk(*self*)	Returns bulk of contents, does *not* include self's bulk.
ContentWeight(*self*)	Returns weight of contents, does *not* include self's weight.
CurrentBulk(*self*)	Returns current bulk of self, including contents bulk. This method assumes self expands from 0 bulk to MaxBulk. In other words, it assumes self is a soft container, like a bag. To create a container (like a box) with a fixed bulk, simply define a method that returns a fixed value instead.
CurrentlyIlluminated(*self*)	Returns TRUE if self is in a lit area, FALSE if not.
CurrentWeight(*self*)	Returns the total weight of self, including contents weight.
DescribeSelf(*self*,*DescriptionArgument*)	*Prints* a description of self based on the argument. Unlike the description methods above, this method actually puts text on the screen. Valid arguments print: Smart - SmartDescribeSelf() Short - SDesc() Long - LDesc() A - ADesc() The - TheDesc() Here - HereDesc() Content - ContentDesc() Name - NamePhrase Adj - AdjectivePhrase Read - ReadDesc() MultiShort - MultiShortDesc() Plural - PluralDesc() Sound - SoundDesc() Odor - OdorDesc()

	Taste - TasteDesc() Feel - FeelDesc() Take - TakeDesc()* Drop - DropDesc()* * From Takeable Item/Fixed Item service
Enter(*self,Object*)	Returns TRUE if object is able to enter self. Checks for a number of conditions and if not successful prints an appropriate message and returns FALSE. This method checks to see if self is openable, self is open, and object can fit inside self.
Insert(*self, Object, Multiple, Silent, Spontaneous)*	Insert self into object. Returns FAILURE if the wrong put preposition was used (Put in table but table has no "in"), or if the Object.*Enter()* method fails. If *Silent* is TRUE then no message is said, otherwise *InsertedDesc()* is called with the *Spontaneous* argument.
IsReachable(*self,Object*)	Returns TRUE if a physical path exists between self and Object. This method is used extensively throughout Universe, from specific object disambiguation, to checking for all manner of senses.
IsVisible(*self,Object*)	Returns TRUE if a visible path exists between self and object.
Leave(*self,Object*)	Returns TRUE if Object can leave self. By default, this method always returns TRUE.
LookDeep(*self*)	Returns FAILURE and complains appropriately if it can't look inside/under/behind/on object, or SUCCESS and prints a *self.LookDesc()* if it can.
Memorize(*self,Object*)	Appends Object to self's Memory list. In other words, allows self to memorize Object.
MoveInto(*self,Container*)	Moves self from its current location to a new location. This function is a primitive, it performs no error checking. Enter() is a better choice for moving objects.
ParentLit(*self,SelfMustBeLit*)	If *SelfMustBeLit* is FALSE returns the outermost container that self would illuminate if self were producing light. In other words, this method returns the outermost container on the light path when *SelfMustBeLit* is FALSE. If *SelfMustBeLit is* TRUE returns the outermost container self *is* illuminating. If self isn't lit this method returns None.
ParentReachable(*self*)	Returns the outermost Parent on the physical path.
ParentRoom(*self*)	Returns the outermost container on the theoretical path. In other words, finds out what room self is in.
ParentVisible(*self*)	Returns the outermost parent on the Visible path. Returns None if there's no light.
Remembers(*self,Object*)	Returns TRUE if self remembers object.
SmartDescribeSelf(*self*)	Prints text on the screen describing the object using the HereDesc() method, SoundDesc() if *BlatantSound* is TRUE, and OdorDesc() if *BlatantOdor* is TRUE. In addition, it causes the current actor to memorize this object.
VerbPrepositon(*self*)	Returns the current verb's *ExpectedPreposition* property, or "inside" if the current verb doesn't have one.
Where(*self*)	Returns *self.Location*. Note that floating methods always replace this function, it is suitable for objects which stay in one place unless moved by the game or the player.

Chapter 7
ClassActor

This class defines "actors". An actor is someone or something the player can talk to or give orders to. Such as "Joe, what is a book?" or "Joe, get book".

Actors are descended from ClassBasicThing, and they have a fixed_item service, which means the player can't take them. You can, of course, override the TakeDesc method if you want the player to take a specific actor (such as a small intelligent animal like a squirrel or ferret that's supposed to be the player's guide).

Properties

The following properties are set by default for actors. You can, of course, override any of these properties as needed.

Bulk	24 (cubic feet, 6x2x2). This default assumes a human build of approximately 6 foot height.
FormatMe	"me". This is the part of speech for "me/him/her". For example "Looks fine to me" but "Looks fine to him".
FormatYou	"you". This is the part of speech for "you". It's "You go north" but "He goes north".
FormatYoum	"you". This is the part of speech for "you/him/her". For example "The monster hits you" but "The monster hits him".
FormatYour	"your". This is the part of speech for "your/his/hers". It's "Your sword" but "his sword".
IsActor	TRUE. Indicates to the parser that objects of this class are actors and can be spoken to.
IsOpen	TRUE. Actors must be open for the inventory process to work properly.
MaxBulk	10 (cubic feet). Assumes the actor can carry 10 cubic feet in bulk. This is approximately 10 boxes 1' x 1'.
MaxWeight	500 (gold pieces, or 1/10 pounds). The actor can carry up to 50 pounds.
NamePhrase	"you". The actor's name phrase defaults to the one used by the player's character.
Weight	1750 (gold pieces, or 1/10 pounds). The actor is assumed to weigh 175 pounds.

Methods

ADesc(*self*)	Returns "yourself". Overrides ClassBasicThing's *ADesc()* method. (Which given the namephrase is "you" would normally say "a you".
Enter(*self,Object*)	Returns *SUCCESS* if object is able to enter actor's inventory, or FAILURE if not. Overrides the *Enter()* method from ClassBasicThing. This method is identical to its overridden ancestor, except for the wording of the complaints.
LDesc(*self*)	Returns "You look about the same as always"
TheDesc(*self*)	Returns "yourself". Overrides ClassBasicThing's ancestor method, which would normally have returned "the you".
Travel(*self,Vector*)	This method returns nothing. It allows an actor to travel in direction *Vector*. *Vector* is always a direction object. You won't normally need to call this function directly, it's normally handled by the parser. If you do call it directly it might look like: *Joe.Travel(North)*. Notice you *don't* put North in quotes, because in this case "North" isn't a string, it's a direction object. See Chapter 9 on ClassDirection (page 35) for an explanation.

Chapter 8
ClassRoom

This class defines rooms and is a direct descendant of ClassBasicThing. A room is a place where an actor can enter. Thus rooms are normally "locations" where the player walks around from one to another. But rooms can also be used as the interiors of vehicles, or djinni bottles, Alice-style rabbit holes, or other more exotic locations.

In actuality a room is simply anywhere the player can be, or to be very picky, any object that can contain the player. This might include items like rubber rafts, balloons or other single "room" vehicles, or even furniture but you'd have to do a bit of coding for these items, the basic room isn't quite so flexible.

Properties

AdjectivePhrase	"". As a rule rooms don't have adjective phrases. This class takes advantage of many ancestor methods, which *do* use the adjective phrase. To let us use them we have to make sure there isn't an adjective phrase, otherwise things would look funny.
IsLit	TRUE. Basic rooms are lit by ambient light (sunlight, for instance, or overhead lighting that's always assumed to be on).
IsOutside	FALSE. This property lets you set which rooms are "outside" (not under a roof) and which ones aren't. It's handy for things like "look at ceiling" or "look at sky".
IsTransparent	FALSE. Rooms are normally opaque. To use transparent rooms you will have to write extra code.
Location	Sky. Rooms (unlike "things") tend not to move. Thus we don't need to give a starting location, we can make their location permanent. The *Sky* object is defined as a basic thing, just to allow us to assign all rooms to a given object. This allows an unobstructed theoretical path between any two objects in the game, which is important for parsing.
Map	Empty. This is the room's map. It's normally set when you create the map portion of your game. Remember you have to define the game map *after* all rooms have been defined.
MaxBulk	32000 (cubic feet). This means the room (to all intents and purposes) is infinite in size, it can hold every object in the game quite easily. For instance, since the player is only 24 cubic feet, a typical room could hold over 1300 humans!
MaxWeight	32000 (gold pieces, 1/4lb pounds). A given room can hold 3,200 pounds. This should be plenty. If needed you can increase this value to a couple of *billion* gold pieces if you need to. Python uses 32 bit integers, not 16. TADS uses 16, which is where the original limit of 32,000 came from. (The max 16 bit signed integer is 32,767).
Open	TRUE. Rooms must be open to allow entry.
Openable	TRUE. Rooms are also openable, by default.
Visited	FALSE. This property is used to keep track of which rooms have been visited and which ones haven't. The scoring routine always adds a room's Value to the player's score the first time they visit a room.

Methods

Enter(*self, Visitor*)	Returns TRUE if *Visitor* can enter the room, FALSE if not. You might think *Visitor* would usually be the player's character, or perhaps another actor. While this is true, this method is *also* used by DropVerb to let the player drop objects in a room. Thus you can override this method to perform all sorts of interesting special effects, from narrow entrances (that restrict bulk) to collapsing bridges (that restrict weight and even kill the player).
FeelDesc(*self*)	Returns "Scrabbling around with your hands reveals nothing useful". Overrides the normal "It feels like an ordinary room to me" response you would have gotten.
FirstView(*self*)	Returns nothing, it's used to increment the player's score by the room value for rooms that haven't been visited yet, and to set the room's visited property to TRUE.
OdorDesc(*self*)	Returns "You don't smell anything.". Note this description will handle different actors correctly. "Joe doesn't smell anything" for Joe, etc.
SmartDescribeSelf(*self*)	Describes the room and its contents. This method shouldn't have to be overridden very much.
SoundDesc(*self*)	Returns "You don't hear anything.". Correctly handles different actors. "Joe doesn't hear anything."

Chapter 9
ClassDirection

Directions are *objects* in PAWS, not prepositions like in TADS. This takes some getting used to if you're used to TADS. Thus you can type "go west" and have the parser handle the translation of "west" into an object the same way "get sword" translates "sword" into an object.

Of course if you type the command "west" this is a *verb*, but we'll explain direction verbs later.

Direction objects have their *AdjectivePhrase* property set to "", and their *Location* property set to None. Otherwise they're straight descendents of ClassBasicThing.

As for methods, only two are changed from ClassBasicThing, first the *Where()* method always returns the current actor's location (so direction objects are always with the current actor) and the *TheDesc()* method simply calls the *SDesc()* method. This lets the parser handle the direction as a normal object and not make the complaints sound weird.

In addition to defining the ClassDirection class for you, we also define 14 direction objects for you as well, 8 for the compass rose directions (north, south, etc), 2 for up/down, two more for in/out and finally two for upstream/downstream.

They are called *North, Northeast, East, Southeast, South, Southwest, West, Northwest, Up, Down, Upstream, Downstream, In,* and *Out.*

Global Object Properties

We also add two additional properties to the *Global* object. These are *in addition* to the ones mentioned already.

The first is *CompassDirectionList*. This gives an array of horizontal direction objects, useful for travel verbs and services.

The second is *DefaultMap*, a fallback of last resort when the player tries to move in a direction you haven't covered in a particular room. Upstream and downstream are perhaps the best examples. Most rooms don't have streams so there's no need for you to worry about them, right?

Chapter 10
ClassMonster

The monster class is nothing more than an actor class with the combat service appended. The only reason this class exists is for code documentation purposes. It makes it much easier to separate potentially hostile actors from non-aggressive ones.

Chapter 11
ClassPlayer

This is the class that the player's character is defined from (or at least an ancestor of that class). It is descended from *ClassMonster* and has changes to properties and methods. We also create an instance of this class called *Me* and assign it to *Global.Player*.

This allows you to override the basic player definition and assign your own. The PAWS convention is that you should *never* refer to *Me*, only to *Global.Player*.

Properties

IsPlural	TRUE. One of the (many) quirks of the English language is that "you" is plural, not singular—even when using the second person singular. Many of the built-in methods of *ClassBasicThing* know how to handle plurals properly.
IsScenery	TRUE. Since the player's character is (usually) the one doing the observing, they should never be included in room descriptions. Scenery is always ignored in room descriptions.
Location	None. The player's location is handled differently.

Methods

ContentsPrefixDesc(*self*)	Returns "You are carrying:" This is the prefix for the inventory command.
EmptyDesc(*self*)	Returns "You are empty handed." This is the complaint if the player isn't carrying anything.
HereDesc(*self*)	Returns "". This is the description of the object in room descriptions.
OdorDesc(*self*)	Returns a snide remark concerning the player's body odor (a string).
SmartDescribeSelf(*self*)	Returns nothing, does nothing. Overrides the ancestor method so no text will be printed when the system calls this function.
SoundDesc(*self*)	Returns "You aren't making any noise."
TasteDesc(*self*)	Returns "You wisely decide against tasting yourself." (a snide remark to the command "Taste me".

Chapter 12
ClassScenery

This class is descended from *ClassBasicThing* and appends the *ServiceFixedItem*. It also sets the *IsScenery* property to TRUE.

Scenery objects are there purely for atmosphere. For example, describe a clearing and you might mention how soft the grass looks, and the leaves scattered around.

It's guaranteed your players will try to cut the grass, mow the grass, get the grass, chew the grass, or somehow interact with the grass when all you had in mind was mentioning the grass in passing.

Scenery is a way to make those dangerously innocent items (grass, trees, sky, ground, etc) able to react to the player's commands easily and without a great deal of coding on your part. With scenery you can simply supply a handful of sensory methods and not worry about inventively stupid players.

Chapter 13
ClassItem

This class is nothing more than *ClassBasicThing* with the *ServiceTakeableItem* appended. It is intended to create objects the player can carry around, such as a rock, a sword, a coin, etc.

Chapter 14
ClassDoor

Doors are defined as *ClassScenery* with *ServiceOpenable* appended. This class is used to create doors. Doors in Universe are always created in *pairs*, one door object in each room. For example to make a door between the living room and kitchen you'd need one door object (*LivingRoomDoor*) for the living room and a second (*KitchenDoor*) for the kitchen. This implementation makes magical doors that teleport the player from one part of the game to another quite simple to make.

Properties

AutomaticOpen	FALSE. Setting the AutomaticOpen option to TRUE allows a player to go through a closed but unlocked door with just "east" rather than "open door; east". It's a courtesy to the player, not a door that opens by itself.
Destination	None. The room this door leads to. Note this is *not* the other door object (see *OtherSide*). If this door were to the kitchen then *Destination* would be set to *Kitchen*.
IsLockable	FALSE. Doors can't be locked by default. If you want a locking door, you should probably use *ClassLockableDoor* instead.
IsLocked	FALSE. Doors aren't locked by default.
IsOpen	TRUE. Doors are open by default.
IsOpenable	TRUE. Doors are always openable.
IsTransparent	FALSE. Doors are generally opaque.
Key	None. Object that unlocks the door. Like the IsLockable/IsLocked properties this is here to ease coding, not because you'd ever set it.
Location	None. The door's location, i.e. *LivingRoom*.
MaxBulk	32000 cubic feet. The biggest object that can pass through the door.
MaxWeight	32000 g.p. (32000 pounds) The heaviest object that can pass through the door.
NamePhrase	'door'. How the door is referred to if you don't override it.
OtherSide	None. The "other side of the door", i.e. the matching door object, for example *KitchenDoor*.

Methods

Close(*self*, *IsSecondTime*)	Closes the door (and the other door object as well). The *IsSecondTime* argument is used to synchronize the two doors so they close together, if you call this method you don't need to pass the argument, it defaults to false. For example *LivingRoomDoor.Close()* will close both the living room and kitchen doors.
Enter(*self*, *Visitor*)	This method allows Visitor to enter the door. This will complain if the door is closed or doesn't lead anywhere. Otherwise, it passes the entry request to the destination room held in *self.Destination*.
Open(*self*, *IsSecondTime*)	Opens the door, and just like *Close* above synchronizes self with *Self.OtherSide*.
SeeThruDesc(*self*)	Returns '' (an empty string). This is the response to *Look Through Door*.

Chapter 15
ClassLockableDoor

This is a ClassDoor with *ServiceLockable* appended. This class is used to create lockable, generally locked and closed doors. Just like regular doors, lockable doors in Universe are always created in *pairs*, one door object in each room. For example to make a door between the living room and kitchen you'd need one door object (*LivingRoomDoor*) for the living room and a second (*KitchenDoor*) for the kitchen. This implementation makes magical doors that teleport the player from one part of the game to another quite simple to make.

Properties

IsLockable	TRUE. If you want doors that aren't lockable use *ClassDoor* instead.
IsLocked	TRUE. As these doors are generally used as barriers they're locked by default.
IsOpen	FALSE. Lockable doors are (naturally) closed by default.
IsOpenable	TRUE. Doors are always openable. Whether the player can do that of course is another story...
LockOnClosing	FALSE. Most doors don't automatically lock when they close, unless they're subject to Murphy's Law...
LocksWithoutKey	FALSE. Most doors require a key to lock them. Notice that one side of a door might be TRUE while the other side could be FALSE.
TransmitLocking	TRUE. If true both *self* and *self.OtherSide* lock together.
TransmitUnlocking	TRUE. If true both *self* and *self.OtherSide* unlock together.
UnlocksWithoutKey	FALSE. Most doors need a key to unlock them, although one side of a door might need a key while the other side does not.

Methods

Close(*self, IsSecondTime*)	Closes the door (and the other door object as well). The *IsSecondTime* argument is used to synchronize the two doors so they close together, if you call this method you don't need to pass the argument, it defaults to false. For example *LivingRoomDoor.Close()* will close both the living room and kitchen doors.
DoorIsLockedDesc()	Returns *"The door is locked"*.
Lock(*self, key, IsSecondTime*)	Returns FAILURE if door doesn't unlock, SUCCESS if it does.
Open(*self, IsSecondTime*)	Opens the door, and just like *Close* above synchronizes self with *Self.OtherSide*.
Unlock(*self, key, IsSecondTime*)	Returns FAILURE if door doesn't unlock, SUCCESS if it does.

Chapter 16
ClassUnderHiderItem/ClassBehindHiderItem

These classes are nothing more than *ClassItem* with the *ServiceRevealWhenTaken* appended. It is intended to create objects that reveal what's under them when moved. Each class has the appropriate *ContainerPrepositionStatic* and *ContainerPrepositionDynamic* set. Other than that, they are no different from *ClassItem*.

Chapter 17
ClassActivateableItem

This class is just *ClassItem* with the *ServiceActivation* appended. It is intended to create objects that can be turned on or off, mainly light sources but any on/off device can be created with just a couple of property changes.

Chapter 18
ClassLandMark / ClassLandmarkMissing

These two classes are simply *ClassScenery* with a modified *Where()* method and a *Landmark* property. For example, if we wanted to create a pine tree and have it appear in every room that has a *HasPineTree* property set to TRUE, then the *Landmark* property would be "PineTree".

When defined with *ClassLandmark* the property must be TRUE for the object to appear, when defined with *ClassLandmarkMissing* the property must be FALSE (but it must still be defined!).

Chapter 19
Services Explained

A *service* is a kind of "miniature class". Services are intended to combine with a *single* class to add a limited set of abilities to the class. For example, adding the *ServiceRevealWhenTaken* service to *ClassItem* creates *ClassUnderHiderItem*, and the only code you have to create is the *SetMyProperties()* method!

With a couple of noted exceptions any services can be combined with any class. Some services come in mutually exclusive pairs, such as *ServiceFixedItem* and *ServiceTakeableItem*, so you obviously can't use them together.

The whole point behind services is to "plug and play". Give an actor *ServiceCombat* and you have a monster. Give *ClassScenery ServicePatrol* and you have a vehicle that moves automatically.

All services require the method *SetServiceProperties()* , even if it contains nothing more than a *pass* statement. This method is used to set any properties the service requires and must be called as part of a class's *SetMyProperties()* method. It should be called after the statement that calls the ancestor class's *SetMyProperties()* method but before overriding existing properties. For example, here's the class definition for *ClassItem()*.

```
class ClassItem(ServiceTakeableItem,ServiceDictDescription,ClassBasicThing):
    """Makes normal items for player to take, drop, etc."""

    def SetMyProperties(self):
        ClassBasicThing.SetMyProperties(self)
        ServiceTakeableItem.SetServiceProperties(self)
        ServiceDictDescription.SetServiceProperties(self)
        self.Descriptions["TakeDesc"]="Taken"
```

Notice in the *class* statement that all services are listed first, then the ancestor class. This is because of the way Python handles multiple inheritance (inheriting properties and methods from multiple classes). The rule is quite simple. When multiple classes are involved, you work left to right, closest first. In other words, you see if the class being defined contains the method. If so, that's the one used. If you don't find it you start tracing backward using *ServiceTakeableItem*. If the method isn't in that service then you start with *ServiceDictDescription* and work backward, and only if you still haven't found the method do you start with *ClassBasicThing* and work backward.

This gives us an easy set of rules to follow when creating classes that use services.

1. Never create a class that inherits from more than one *Class*. For example, don't create a class that uses both *ClassBasicThing* and *ClassActor*!
2. Always list services *first*, as in our *ClassItem* example. This means that service methods always override the methods from the base class. This is how *ServiceDictDescription* is able to replace *LDesc()* for example.
3. For simplicity's sake never include two services with the same methods if you can help it. For example, never include both *ServiceFixedItem* and *ServiceTakeableItem* in the same class! The result will be that the methods from the service listed first will be used; the methods from the second service will be ignored. Not only will this confuse anyone reading your code, it's wasteful in terms of computer memory and processing time as well.
4. If you want to create your own services please remember that services have neither ancestors nor descendents, they are never part of an inheritance tree by *design*. The primary reason for this is to prevent confusion as much as possible when dealing with multiple inheritances. This is also the reason for rule #1.

Chapter 20
ServiceActivation

This service allows devices to be activated and deactivated. This is generally used for light sources but can be used for any device that is switched on or off, with or without a tool.

Properties

ActivatePassivePhrase	"You light the lamp."
ActivateSpontaneousPhrase	"The lamp lights up".
ActivationProperty	"IsLit". The name of the TRUE/FALSE property that this service manipulates. By default it's "IsLit". Notice this property is a *string*.
AlreadyActivatedPhrase	"The lamp is already lit."
AlreadyDeactivatedPhrase	"The lamp is already out."
DeactivatePassivePhrase	"You douse the lamp."
DeactivateSpontaneousPhrase	"The lamp goes out."
RequiredActivationTool	None. The object required to activate self. For example, if self were a candle then *self.RequiredActivationTool* might be *Match*.
RequiredDeactivationTool	None. The object required to deactivate self. For example, you might use a match to activate (light) the candle, but you'd use a candle snuffer to douse it. Or you might not need any tool to douse the candle.
MaxLifeSpan	32000. The maximum number of turns the device can operate before being refueled/recharged/whatever.
RemainingLifeSpan	32000. How much operating time (in turns) is left in the device.

Methods

Activate(*self, Multiple, Spontaneous, Silent*)	Activates the device. Makes all the appropriate checks and complains. Returns SUCCESS if self could be activated, FAILURE if not.
Deactivate(*self, Multiple, Spontaneous, Silent*)	Deactivates the device. Makes all the appropriate checks and complains. Returns SUCCESS if self could be deactivated, FAILURE if not.
DrainLife(*self*)	Reduces *self.RemainingLifeSpan* by 1, calls *self.Deactivate()* when the remaining life reaches 0.
AlreadyActivatedDesc(*self*)	Returns *self.AlreadyActivatedPhrase*.
AlreadyDeactivatedDesc(*self*)	Returns *self.AlreadyDeactivatedPhrase*.
LifeRemaining(*self*)	Returns *self.RemainingLifeSpan*.
RequiresToolDesc(*self*)	Returns *"You'll need something to do that with."*
WrongToolDesc(*self*)	Returns *"You can't do that with a rock."*

Chapter 21
ServiceDictDescription

This service allows the game author to avoid having to create classes simply to add common description methods like LDesc() and the other sensory description methods. Instead a single class can be used to create multiple objects with different descriptions (such as rooms) without having to create a class for every room.

Universe uses this service to create rooms, scenery, and items. For any object using the *VarDesc()* method this service is part of the class.

Properties

Descriptions	A dictionary of descriptions, these replace many of the more frequently replaced description methods, like Ldesc(), Sdesc(), etc. See the methods below to see which ones are replaced by descriptions.

Methods

DefaultDescriptions(*self*)	Places the default descriptions for the service into the *Descriptions* dictionary.
FeelDesc(*self*)	Replacement of feel description that uses the Descriptions dictionary instead.
LDesc(*self*)	Replacement of long description that uses the Descriptions dictionary instead.
OdorDesc(*self*)	Replacement of odor description that uses the Descriptions dictionary instead.
SetDesc(*self, Key, Value*)	An easy method to place saring descriptions into the *Descriptions* dictionary. The *Key* is the name of the description being replaced (one of the methods mentioned), minus *"Desc"*. Thus for LDesc the *Key* is *"L"*. The value is the description to be put in the dictionary, you may use curly brace expressions in it.
SoundDesc(*self*)	Replacement of sound description that uses the Descriptions dictionary instead.
TasteDesc(*self*)	Replacement of taste description that uses the Descriptions dictionary instead.

Chapter 22
ServiceOpenable

This service allows devices to be opened or closed. This is generally used for containers but can be used for any object you can open, such as a locker.

This service doesn't require additional properties beyond those found in *ClassBasicThing*.

Methods

AlreadyClosedDesc(*self*)	Returns *"The chest is already closed."*
AlreadyOpen(*self*)	Returns *"The chest is already open."*
Close(*self, Multiple, Silent, Spontaneous*)	Returns TRUE if the object closes, FALSE if it doesn't. If *Silent* is TRUE then no description is primed on a successful close. *Multiple* is passed from the verb, it determines if this object is part of a list of objects being closed. This service ignores *Multiple*, but including it allows you to create a new service that will honor *Multiple* without disrupting the system.
CloseDesc(*self, Multiple, Spontaneous*)	Returns *"The chest closes."* if *Spontaneous* is TRUE, or *"You close the chest."* If *Spontaneous* is FALSE.
Open(*self, Multiple, Silent, Spontaneous*)	Returns TRUE if the object opens, FALSE if it doesn't. If *Silent* is TRUE then no description is primed on a successful open. *Multiple* is passed from the verb, it determines if this object is part of a list of objects being opened. This service ignores *Multiple*, but including it allows you to create a new service that will honor *Multiple* without disrupting system.
OpenDesc(*self, Multiple, Spontaneous*)	Returns *"The chest opens."* if *Spontaneous* is TRUE, or *"You open the chest."* If *Spontaneous* is FALSE.
UnopenableDesc(*self*)	Returns *"You can't open the rock."*

Chapter 23
ServiceLockable

This service allows objects (usually doors) to be locked.

Properties

IsLocked	FALSE. Set this to TRUE if the object should be locked. For example, *TreasureChest.IsLocked = TRUE*.
LocksWithoutKey	TRUE. Set this to FALSE if you want a key to be required for locking the object.

Methods

AlreadyLockedDesc(*self*)	Returns *"The chest is already locked."*
AlreadyUnlockedDesc(*self*)	Returns *"The chest is already unlocked."*
Lock(*self, key, Silent, Spontaneous*)	Returns TRUE if the object locks, FALSE if it doesn't. If *Silent* is TRUE then no description is printed on a successful lock. *Key* is the object to lock self with.
LockDesc(*self, Multiple, Spontaneous*)	Returns *"The chest locks."* if *Spontaneous* is TRUE, or *"You lock the chest."* if *Spontaneous* is FALSE.
NeedAKeyDesc(*self*)	Returns *"You need a key to do that."*
Unlock(*self, key, Silent, Spontaneous*)	Returns TRUE if the object unlocks, FALSE if it doesn't. If *Silent* is TRUE then no description is printed on a successful unlock. *Key* is the object to unlock self with.
UnlockDesc(*self, Multiple, Spontaneous*)	Returns *"The chest unlocks."* if *Spontaneous* is TRUE, or *"You unlock the chest."* if *Spontaneous* is FALSE.
WrongKey(*self, Key*)	Returns *"This key doesn't work with this chest."*

Chapter 24
ServiceRevealWhenTaken

This service allows objects to drop their contents when moved. It's used to create "hider" items, items which basically contain what they're hiding until moved or taken, then (silently) drop their contents so it appears they were actually sitting in front (or on top) of the items all the time.

Methods

Take(*self*)	Returns *SUCCESS* if the object can be taken, *FAILURE* if it can't. In addition it drops self's contents and describes them appropriately.

Chapter 25
ServiceTakeableItem

This service allows objects to be picked up and dropped. It requires no properties aside from those found in *ClassBasicThing*.

Methods

Drop(*self*, *Multiple*)	Returns SUCCESS if the object can be dropped, FAILURE if it can't.
DropDesc(*self*, *Multiple*)	Returns *"Dropped"* unless this object is part of a multiple object drop, in which case it returns *"rock: dropped"*.
NotCarryingDesc(*self*)	Returns *"You aren't carrying that."*
Take(*self*, *Multiple*)	Returns SUCCESS if the object can be taken, FAILURE if it can't.
TakeDesc(*self*, *Multiple*)	Returns *"Taken"* unless this object is part of a multiple object drop, in which case it returns *"rock: taken"*.

Chapter 26
ServiceFixedItem

This service prevents an item from being taken. It complains appropriately if the player tries to take or drop self. It requires no properties other than those found in ClassBasicThing.

Methods

Drop(*self*, *Multiple*)	Returns FAILURE since you can't drop an object you aren't carrying in the first place.
DropDesc(*self*, *Multiple*)	Returns *"You aren't carrying that."* unless this object has a *Description* dictionary, in which case it returns the dictionary entry for "DropDesc".
Take(*self*, *Multiple*)	Returns FAILURE because you can't take a fixed object.
TakeDesc(*self*, *Multiple*)	Returns *"You can't take that!"* unless this object has a *Description* dictionary, in which case it returns the dictionary entry for "TakeDesc".

Chapter 27
ClassBasicVerb

This object is descended from PAWS ClassBaseVerbObject, and all Universe verbs are made from it. It implements only two methods of its own, *SpecificDisambiguation(self)* and *SanityCheck(self)*.

Specific Disambiguation

This class only exists to implement specific disambiguation. In other words, to help the parser figure out exactly which object the player means when he types "get key" and there are 4 different keys in the game. You can override this method on a verb-by-verb basis, but most of the time you'll never need to.

As implemented in Universe specific disambiguation allows the parser to check ambiguous object references (like our example key) by process of elimination. If all objects referred to by the noun ("key") are eliminated an appropriate error message is printed. Most of the time you'll never have to provide specific reason or error methods at all!

The disambiguation routine applies the following checks, in sequence, to discard objects until only the correct one is left.

- Is object addressed as an actor really an actor?
- Does verb have a list of allowed direct/indirect objects, and if so is the object on it? This test always passes if the verb has no objects in the direct/indirect object list.
- Does player know about object yet? More accurately, does the player's *character in the game* know about it yet?
- Is the object visible?
- Is the object reachable?

If any of the above tests fail the object is discarded. If all objects for a given noun are discarded the appropriate error message is printed.

If you want details about how this routine works, consult the source code for *ClassBasicVerb* and the PAWS function *DisambiguateListOfLists()*.

Sanity Check

This function basically verifies that the player has light to see how to do the command, or that the command can succeed in the dark. If either condition is true it returns *SUCCESS*, otherwise it complains "It's too dark to see how".

Chapter 28
Verbs

Verbs in *Universe* are intended to be simple switching gates, they basically call the direct object's *verb method* and that's all. For example, if the player types *Drop rock* then the *DropVerb* basically calls *Rock.Drop()*, passing a few optional arguments. This allows new verbs to be created very easily and passes the responsibility of executing an action to the object where it really does belong.

Because of this instead of listing each verb in its own chapter, we'll simply create a table listing the most important and frequently changed aspects of the verbs. If you want details about any particular verb, just look at the verb's code.

Verb	Direct Objects	Indirect Objects	Ok in Dark	Verbs	Prepositions	Class
AgainVerb	None	None	Yes	G, Again	None	ClassSystemVerb
ClimbVerb	Multiple	None	Yes	Climb	None	ClassGoVerb
CloseVerb	Multiple	None	Yes	Close	None	ClassCloseVerb
DebugVerb	None	None	Yes	Debug	None	ClassDebugVerb
DownstreamVerb	None	None	Yes	Downstream, ds	None	ClassTravelVerb
DownVerb	None	None	Yes	Down, d	None	ClassTravelVerb
DropDownVerb	Multiple	None	Yes	Put, set, throw	Down	ClassDropVerb
DropVerb	Multiple	None	Yes	Drop, release	None	ClassDropVerb
EastVerb	None	None	Yes	East, e	None	ClassTravelVerb
ExamineVerb	Multiple	None	No	Examine, inspect, x		ClassLookAtVerb
ExtinguishWithVerb	Multiple	One	Yes	Extinguish, douse	With	ClassDeactivateVerb
ExtinguishVerb	Multiple	None	Yes	Deactivate, extinguish, douse	None	ClassDeactivateVerb
FeelAroundVerb	Multiple	None	Yes	Feel	Around	ClassFeelVerb
FeelVerb	Multiple	None	Yes	Feel, touch		ClassFeelVerb
GoToVerb	Multiple	None	Yes	Go, walk, run, move	To	ClassGoVerb
GoTowardVerb	Multiple	None	Yes	Go, walk, run, move	Toward	ClassGoVerb
GoVerb	Multiple	None	Yes	Go, walk, run, move	None	ClassGoVerb
HangOnVerb	Multiple	One	Yes	Hang	On	ClassInsertVerb
HelloThereVerb	Optional	None	Yes	Hi, hello	There	ClassHelloVerb
HelloVerb	Optional	None	Yes	Hi, hello		ClassHelloVerb
InventoryVerb	None	None	Yes	Inventory, inven, i		ClassInventoryVerb
InVerb	None	None	Yes	In, enter, ingress		ClassTravelVerb
LightVerb	Multiple	None	Yes	Light, activate		ClassActivateVerb
LightWithVerb	Multiple	One	Yes	Light	With	ClassActivateVerb
ListenToVerb	Multiple	None	Yes	Listen	To	ClassListenToVerb
ListenVerb	None	None	Yes	Listen		ClassListenVerb

LockVerb	One	None	Yes	Lock, latch, hook	None	ClassLockVerb
LockWithVerb	One	One	Yes	Lock	With	ClassLockWithVerb
LookAroundVerb	None	None	No	Look, l	Around	ClassLookVerb
LookAtVerb	Multiple	None	No	Look, l	At	ClassLookAtVerb
LookBehindVerb	None	One	No	Look, search	Behind	ClassLookDeepVerb
LookInsideVerb	None	One	No	Look, search	In, inside, into	ClassLookDeepVerb
LookOnVerb	None	One	No	Look, search	On	ClassLookDeepVerb
LookUnderVerb	None	One	No	Look, search	Under	ClassLookDeepVerb
LookVerb	None	None	No	Look, l, gaze	None	ClassLookVerb
NortheastVerb	None	None	Yes	Northeast, ne	None	ClassTravelVerb
NorthVerb	None	None	Yes	North, n	None	ClassTravelVerb
NorthwestVerb	None	None	Yes	Northwest, nw	None	ClassTravelVerb
OpenVerb	Multiple	None	Yes	Open	None	ClassOpenVerb
OutVerb	None	None	Yes	Out, outside, exit	None	ClassTravelVerb
PickUpVerb	Multiple	None	Yes	Pick	Up	ClassTakeVerb
PutBehindVerb	Multiple	One	Yes	Put, place, hide, set	Behind	ClassInsertVerb
PutInVerb	Multiple	One	Yes	Put, place, hide, set	In, into, inside	ClassInsertVerb
PutOntoVerb	Multiple	One	Yes	Put, place, pile, stack, set	On, onto	ClassInsertVerb
PutOutVerb	Multiple	None	Yes	Put	Out	ClassDeactivateVerb
PutUnderVerb	Multiple	One	Yes	Put, place, hide, set	Under, underneath, beneath	ClassInsertVerb
QuitVerb	None	None	Yes	Quit	None	ClassQuitVerb
ReadVerb	Multiple	None	No	Read	None	ClassReadVerb
RestoreVerb	None	None	Yes	Restore	None	ClassRestoreVerb
SaveVerb	None	None	Yes	Save	None	ClassSaveVerb
SayVerb	Optional	None	Yes	Say	None	ClassSayVerb
SmellVerb	Optional	None	Yes	Smell, sniff	None	ClassSmellVerb
SoutheastVerb	None	None	Yes	Southeast, se	None	ClassTravelVerb
SouthVerb	None	None	Yes	South, s	None	ClassTravelVerb
SouthwestVerb	None	None	Yes	Southwest, sw	None	ClassTravelVerb
TakeInventoryVerb	None	None	Yes	Take	Inventory	ClassInventoryVerb
TakeStockVerb	None	None	Yes	Take	Stock	ClassInventoryVerb
TakeVerb	Multiple	None	Yes	Take, get, remove, steal	None	ClassTakeVerb
TasteVerb	Multiple	None	Yes	Taste, lick	None	ClassTasteVerb
TurnOffVerb	Mutiple	None	Yes	Turn	off	ClassDeactivateVerb
TurnOnVerb	Multiple	None	Yes	Turn	On	ClassActivateVerb
UnlockVerb	One	None	Yes	Unlock	None	ClassUnlockVerb
UnlockWithVerb	One	One	Yes	Unlock	With	ClassUnlockWithVerb
UpstreamVerb	None	None	Yes	Upstream, us	None	ClassTravelVerb
UpVerb	None	None	Yes	Up, u, ascend	None	ClassTravelVerb
WestVerb	None	None	Yes	West, w	None	ClassTravelVerb