Design Documentation for MOZ (Moo in OZ).

Robin Lee Powell

This manual is for MOZ (MOO in Oz) version 1.0.

Table of Contents

This is the design documentation for MOZ (Moo in OZ). MOO is Mud Object Oriented. MUD is Multi-User Dungeon or Dimension. In general, a MUD is a multi-user text-based virtual environment. For information on MUDs in general, see The MUD Resource Collection or your local search engine. For information on MOOs, see The MOO-Cows FAQ.

Oz is a multi-paradigmatic language that happens not to suck. See The Main Mozart-Oz Page.

Note that the requirements document is also incorporated herein, and that this document is changing in response to actual code decisions, and hence is not strictly a design document. Oh well.

1 Requirements

An informal requirements set for Moo in OZ. Note that these are in no particular order. They are anchored for reference in those parts of the design document that relate to them.

1. multiple muds can be joined together transparently
2. ability to make as many changes on the fly as possible
3. maximum simplicity for building and coding
4. not too slow or too memory-hoggy
5. number of connections limited only by physical issues (bandwidth, memory, etc)
6. robust access control, within the MOZ, of objects, of internal code, of objects characteristics, of processes, etc.
7. ability to handle various (spoken) languages
8. high potentiol for user enjoyment
9. fault tolerance
10. ease of debugging
11. user can see general information, including class code, about objects in the MOZ
12. ease of testing
13. ease of modification on the part of new admin, programmers, and builders

2 Security Overview

Capability security is a security model that is most likely entirely different than what you're used to. It is simple and elegant and fast. The main drawbacks are that it has the potential to make it much easier for users to do incredibly stupid things, but that really depends on how the system is set up, and that it is very hard to bootstrap. The latter is solved by persistence, see See Section 8.4 [Persistence], page 23. The former is solved in MOZ by simply not allowing the transfer of capabilities that are innapropriate.

OK, time to step back a bit and explain more about capability security as a concept. The type of security that you are most probably used to I'm going to call ACL (Access Control List) security. The way this basically works is that if you want to, say, open a file, somewhere buried in the operating system is a test that compares your user name to a list of people allowed to open the file. Or if you're in a MUD and want to pick up an object, same thing: somewhere an if statement tests if you should be allowed to do that.

The basic idea with ACLs is that you can always find what it is that you want to access. File trees have ways to get listings of all the file names, which you can then pass to an open command of some kind. Rooms have a list of all the objects in them, which you can then pass to a get command of some kind.

Capability security is completely different. Capability security is based on the idea of a key to a vault: you either have the key, or you don't. If you don't have the key, you can't open the vault (at least, not without a whole lot of time and effort; good cryptography, which MOZ as of this writing only sort of uses, can make the time involved into the trillions of years).

Better still, in most cases in a capability system you can't even *see* the vault (again, without a lot of time and effort and access to the data in some kind of raw form, such as access to the disk itself). If you can't point to the vault or see it or touch it, it's ridiculous to talk about opening it: you don't have the *capability* to do so at all (hence the name of this type of security).

Basically, a capability is a reference to some kind of operation on an object (i.e. generally it's a reference to a procedure). If you don't have the capability, and you can't forge the capability (because you can't see the object it's acting or you can't guess the capability's name/secret/key/whatever), then you can't access that aspect of the object.

Note that the key analogy holds in other ways: you can copy capabilities, and you can give them to other objects. This is what leads to the first drawback mentioned above: players can quite easily "give away the homeworld" Babylon 5 Quotes.

Some effort is made to make this difficult (note that requiring actual Oz programming to transfer a capability is considered 'difficult'), but it's part of the nature of the system that this cannot be stopped.

On the other hand, capabilities can be revoked. In fact, being able to tell an object to revoke a capability is, itself, a capability in MOZ!

Which leads to one of the other nice things about capabilities: they're arbitrarily extendable. You want to make another one, you just program it in and pass it on. Quite trivial, whereas, in general, ACL-based systems require reworking of at least some of the system code to add a new ACL.

Furthermore, you can pass around a procedure that has a capability inside it that it will only use once. Since procedures can't be opened up or de-compiled in Oz, this is quite safe.

An important aspect of capabilities is that, unlike ACL systems, an object can, by default, do *nothing*. At least, nothing that involves other objects. This is very important: hacks of ACL systems often revolve around tricking the ACL into not activating, or believing a user is someone they're not, or something. This simply isn't possible in a capability system: there's nothing to trick. To even *ask* for a capability you don't have requires a capability! This motivates the next section, which defines an initial set of capabilities.

For more information, see The Three Parts of Security, Introduction To Capability Based Security, and An Ode to the Granovetter Diagram.

Note that almost all security in MOZ, is done from an in-language, in-db perspective. Some aspects of the security rely on ex-DB code (the ActiveObject wrapper in particular), but these are the exception rather than the rule.

If it's important enough to you to break into a MOZ that you start eavesdropping transmissions between MOZs to figure out how Oz encodes its name information, you're welcome to hack the thing as far as I'm concerned. Go get laid or something, you're a loser.

3 One-Instance Classes

There are a number of classes which are instantiated exactly once per server, i.e. there is exactly one object of each of these classes on each server. These objects are used to store and manage server-wide persistent values. For example, the Storage object keeps track of all the files that are used to store objects to disk.

Because of this, I will often refer to the class and the object interchangeably; the Storage class and the Storage object are largely the same thing in practice, as there is only ever one Storage object made from the Storage class.

The one-instance classes are listed below, along with a brief description of the types of data they store.

'`The Storage Object`'
: The Storage object stores information about the disk files used to store objects and classes. It stores associations between Oz names and file numbers and object wrappers and other data. Chapter 6 [Storage Issues], page 16

'`The Server Object`'
: The Server object stores a list of players and their associated passwords.

'`The LanguageStrings Object`'
: The LanguageStrings objects stores localized strings for everything that the core system classes need to output to the user. This is so that people wishing to localize MOZ to a new language don't need to go looking for strings all over the place.

'`The Help Object`'
: The Help object stores all the basic help information of the MOZ.

4 Ex-DB Design

The phrase "Ex-DB" refers to the idea that in MUD code, anything you can change from within the MUD, without having to reset it or something, is "In-DB", as in "in the database". By extension, anything that you cannot change from within the MUD is "Ex-DB", or "out of the database".

MOZ is very different from other MUDs. The server, as such, is nearly non-existent; it's just a tiny bootstrap module (see [R2], page 2,see [R7], page 2), plus a few accessory modules. This is because all things in Oz, including classes and objects, are first-class values. Even the the class of an object can be changed on the fly.

4.1 The moz Program

The main program binary, moz, performs the following functions:

1. Compiles MozBase.class using MozCompilation
2. Compiles Storage.class using MozCompilation
3. Creates a new, empty Storage object by hand
4. Uses Storage to load itself into itself
5. Starts the Storage object
6. Asks the Storage object where to find the Server object (i.e. what the Server object's file number is).
7. Tells storage to load the server object
8. Starts the Server object
9. Holds until the server is shut down
10. Ask Storage to write everything to disk
11. Writes the Storage object to disk by hand.

4.2 The new_moz Program

The new_moz program creates a bare, clean, and very boring MOZ instance. It's just like the moz program except that it creates inital instantiations of all of the one-instance classes, and it doesn't load the Storage object from disk.

4.3 The Parsing Module

The Parsing module is a set of monadic parser combinators, which can be used to parse any CFG in a pretty simple, straight-forward way. The may be more expressive than CFGs with a bit of work, I don't know. The code is basically a re-write of the code at Monadic Parser Combinators, which is a lovely tutorial on monadic parsing.

This code will eventually be on MOGUL, when I manage to get around to it.

4.4 The EnhancedString Module

This is just a re-packaging of Duchier's MOGUL String Package, because it's nice to have the better string handling capabilities around.

4.5 The Active Object Wrapper

The active object wrapper that MOZ uses is one of the more complicated pieces of code in the MOZ, as well as the piece that may seem oddest to users of other object oriented langauges. Also, it is the largest piece of ex-db code.

The active object wrapper is actually a procedure that is passed an object. It returns an object record with full capabilities. An object record consists of a wrapper procedure, the objects Oz name, and a capability set. All the wrapper procedure does is pass its one argument to a port (an Oz port, *not* a TCP/IP port).

It's what's at the other end of that port that's interesting. A thread is created to process the stuff passed in the port. Note that this means that each object has exactly one thread associated with it, and this thread is the only one that ever gets to affect the object directly. This make things much easier as far as multi-threading issues go, as no locking needs to be done within object methods.

This wrapper also gives insultion: no-one ever sees the object directly except this thread, and this is what allows the capability security system to work.

Also, the thread can do other processing besides just calling the method it has been passed. The most important part of this processing is capability processing. Basically, what's passed to the port is not the method name, it's a record with an Oz name for a label. This label is used to key a dictionary, and if a method name associated with it is found, that method is called, if not, an error is returned. For more information, see See Section 8.3 [Capability Implementation], page 22.

Another, much cooler, thing that the wrapper thread does is handle upgrade requests. upgrade is a virtual method, in that it is called just like any other method in the wrapper, using an Oz name capability, but it doesn't exist in the object, rather it exists in the wrapper. An upgrade call takes a class name and actually swaps out the object it's wrapping for one of that class! It calls `toRecord` and `fromRecord` to make sure information is preserved, and lets the Storage object know the new class name. Then it just goes back to processing the port like nothing happened, and from the point of view of every other object in the mud, nothing did!

There are a couple of other miscellaneous things the wrapper does. In particular, it inserts capability information into a `toRecord` call and extracts it from a `fromRecord` call.

When a new object is created, the capability dictionary is built using the class's methodList feature, with the addition of the special capability 'upgrade', and a bunch of new Oz names. In many cases, these are then overwritten by a `fromRecord` call.

The object itself will often need to know about its wrapper and capabilities, so that it can inform other objects. This is handled by a method on every object in the MOZ called activeObjectInform, which takes those two pieces of information as arguments, generally to store on attributes. This method is called every time wrapper information is changed.

The Storage object is also informed of this information at the same time.

See Section 5.1 [General Class Issues], page 9.

4.6 The MozCompilation Module

The MozCompilation module exports a single procedure, `Compile`, which takes the file name to compile and the environment to compile it under. If the file contains only an expression,

such as a single functor, that is returned. The environment is given as a list of 2-tuples. The first part of each 2-tuple is an atom representing the variable name under which that aspect of the environment will be visible in the compilation. The second part is the actual value being made so visible. Note that if one of the variables passed in to the compilation is unbound, binding it in the compilation will effectively work as output.

If you didn't understand that, that's probably fine; this isn't a user-servicable part of the code, and I don't really even understand it myself.

4.7 Object References

Dealing with the active object wrapper directly is difficult, due to having to keep track of a seperate list of capabilities, so we wrap it in an Oz object. This wrapping is done, most of the time anyways, by using the procedure *MakeObjectReference*, supplied by ActiveObject.oz.

For arguments, it takes an ActiveObject wrapper, a record of capabilities, and the Oz name of the object (more for convenience than anything else), and returns an object which runs the wrapper but also implements the methods `"wrapper"`, `"capabilities"`, and `"ozName"`, which simply return the information passed to them in both the first argument and the argument with the same label as the method name.

5 Class Definitions

5.1 General Class Issues

5.1.1 The `start` Method

Every class has a `start` method, which is run at the end of the process of the object being loaded in by Storage. Note that this is run *any* time the object is loaded, including as a result of upgrade calls. Therefore, `start` must not assume, well, *anything*. For example, the `start` method on a wandering puppet, that starts its wandering from room to room, must not assume anything about what room it starts in.

Note that any initialization of anything that is not completely permanent should happen using the `start` method, not the `init` method.

5.1.2 The `init` Method

Every class has an `init` method, which is run by Storage as part of the object creation and wrapping process. The `init` method, as a minimum, takes an object record for Storage and LanguageStrings and takes the object's own Oz name.

NOTE: The `init` method should only be called once for the *entire* lifetime of the object. Anything that is not absolutely permanent should not be passed to `init`. Also, an `init` record can be passed to createObject on Storage, but be aware that Storage will change the name of the record to `init` and tack on three arguments at the beginning for the Oz name and the Storage and LanguageStrings object references.

5.1.3 The `toRecord` Method

Every class has a `toRecord` method, which returns a record containing all the aspects of the object that must be persistent (see Section 6.1 [Loading And Unloading], page 16. Note that reference to objects *must* be converted to their appropriate Oz names before being made part of the record, because the actual object code cannot be stored on disk. The names in question are only used for persistence purposes, and knowing them should give no privilege[1].

5.1.4 The `fromRecord` Method

Every class has a `fromRecord` method, which takes the output of `toRecord` and applies the information therein to the object (see Section 6.1 [Loading And Unloading], page 16). Oz names representing objects are turned back into object references as part of this process.

5.1.5 `toRecord`, `fromRecord`, and `exports`

The `toRecord` method uses a feature called `exports`, which is a list of 2-tuples of the format Name#Type, where both Name and Type are atoms. The former is the attribute name to be saved, and the latter is the type of the attribute. Note that these types are specific to MOZ, and not Oz itself. The Type information is used to determine when special processing

[1] It is possible that the current method of doing `toRecord`, in that it is defined on MozBase, will turn out to be a security hole, because objects can lie about themselves. If so, it should be moved onto ActiveObject.

is needed to be able to safely save an attribute to disk. For example, `toRecord` uses the Type field to look for object references, which need special processing.

It is very important that the `export` list contain only the names of attributes that contain non-stateful data, or else the save will fail. Stateful data is data that is specific to a particular invocation of a running program, such as open file handles. Oz programmers also refer to stateful data as 'resources'.

Simple records, atoms, Oz names, and procedures are all safe.

Threads, file handles, and any other value that isn't guaranteed to be persistent, are all unsafe to save. Any attribute that stores this form of data either must not be named in the `export` list, or there must be special code in `toRecord` to deal with that Type, such as there is for the type called `objectRef`, which is for references to objects.

There is also an analogous `featExports` list for saving the value of features. Since features are write-once values, an object that has a featExports list should only have fromRecord called on it once. Any subsequent calls once the features have been initialized will fail.

5.1.6 Recognized `exports` Types in `toRecord` and `fromRecord`

Type values that currently mean somethng: `dict`, `objectRefDict`, `notPersistent`, and `objectRef`. Anything else will be treated as in no way special. This means if it has non-persistent data, an error will result.

The standard `toRecord`/`fromRecord` methods have a Type `objectRef` that has a fairly specialized format. These attributes are used to refer to other objects in the MOZ. It expects the attribute to which it refers to be a record with the field names `ozName`, `wrapper` and `capabilities` which point to the Oz name of the object, the active object wrapper procedure and the capabilities that this object has on the object being referenced. The list of capabilities is a list of methods on the object pointed to by the `objectRef` that the holder of the `objectRef` can use.

Note that this means that an object cannot save persistently information about an object whose Oz name it cannot retrieve, but this should not be a limitation in general.

The Type `notPersistent` is used to pass non-persistent data, such as the output port for a Player object to send data to the player, through upgrade calls. It is not used for `toRecord` calls for data destined to be writen to disk, as an error will result when trying to store that type of data (as the concept makes very little sense).

5.1.6.1 Examples

Note that these examples are *not* valid MOZ code, or even Oz code really. They are just intended to give a bit of a visual idea of how `toRecord`, `fromRecord`, `exports` and `objectRef`-Type attributes interact.

- A simple object: DrinkingGlass173

  To store DrinkingGlass173 to disk when it is not being used, we use `toRecord`. In turn, `toRecord` uses `exports` and the `objectRef` Type. This is what part of the object it might look like:

```
exports list:
    name#string
```

```
    description#string
    storageRef#objectRef
```

Note that `name` is not a specially recognized attribute in Oz or MOZ; it is user-defined, and objects need not have it. In MOZ, most objects do in fact have a `name`, but it is not required.

An `objectRef` is an Oz object itself, wrapping the ActiveObject wrapper for easier use. The information it contains consists of the wrapper for the object it refers to, the object's ozName, and some capabilities on that object, which we will represent like so:

```
    storageRef:
wrapper
ozName
capabilities: capabilities(
    createObject
    objectRefFromRecord
    getClass
    upgradeObject
)
```

Note that every object that gets save to disk has at least one entry in its `exports` list that is of Type `objectRef`: `storageRef`, which is used to allow the object to create other objects and load class code and such.

- A complex object: "Robin's Room"

  To store "Robin's Room" we use the same `toRecord` methed. This is what part of the object it might look like:

```
exports list:
    name#string
    description#string
    storageRef#objectRef
    contents#objectRefDict

    .
    .
    .

    Contents: contents(
blueChair
redChair
drinkingGlass173
    )

    .
    .
    .

    BlueChair: objectRef(
wrapper
```

```
ozName
capabilities: capabilities(
    getName
    getDescription
)
    )
```

5.1.7 methodList

Every class has a `methodList` feature that is a list of all the method names that are used outside the class. This is used by the active object wrapper, and is safe since nothing but the active object wrapper ever talks directly to the object anyways. Note that the `methodList` feature *should not* include upgrade, as this is added in by the active object wrapper.

All objects are dynamically upgradeable a la A dynamically-extensible server[2]

5.2 The Class Hierarchy

This is a basic outline of which classes descend from which other classes.

- Section 5.3 [class MozBase], page 13
  - Section 5.4 [class Storage], page 13
  - Section 5.5 [class Server], page 13
  - Section 5.6 [class Described], page 13
    - Section 5.9 [class Location], page 13
      - Section 5.13 [class Container], page 14, also a child of Section 5.8 [class Mobile], page 13
      - Section 5.10 [class Player], page 14, also a child of Section 5.8 [class Mobile], page 13
      - Section 5.16 [class Terminus], page 15
    - Section 5.7 [class Located], page 13
      - Section 5.8 [class Mobile], page 13
        - Section 5.13 [class Container], page 14, also a child of Section 5.9 [class Location], page 13
        - Section 5.10 [class Player], page 14, also a child of Section 5.9 [class Location], page 13
          - Section 5.11 [class Wizard], page 14
          - Section 5.12 [class Puppet], page 14
        - Section 5.19 [class Control], page 15
        - Section 5.20 [class ClassControl], page 15
      - Section 5.14 [class Exit], page 14
        - Section 5.15 [class Gate], page 14
  - Section 5.17 [class Connection], page 15
  - Section 5.18 [class Parser], page 15

[2] Note that this also makes them stationary, which may cause problems later (although I don't think so) and means that what's actually being passed around are procedures, not objects. For more information, see See Section 4.5 [The Active Object Wrapper], page 7.

5.3 class MozBase

An extension of BaseObject, and an ancestor of *every* other class. MozBase has an 'ozname' feature, and the `init` call initializes 'ozname' to `{NewName}`, as well as storing a reference to the Storage class. MozBase also has a method 'ozname' that returns this name. All of the methods listed in Section 5.1 [General Class Issues], page 9 are first defined in MozBase.

MozBase will not usually be directly instantiated.

5.4 class Storage

class Storage deals with the saving and loading of classes and objects, including doing Active Object wrapping (see Section 4.5 [The Active Object Wrapper], page 7). It also provides support for mapping between file names, Oz names, and object references (see Section 6.2 [Associations], page 16).

In fact, the Storage class does an awful lot of things. It passes initial capabilities to the object and remembers which class each object belongs to (see Section 8.1 [Initial Capabilities], page 21). It stores capabilitites on each object for purposes of Wizardry (see Section 8.5 [Wizardry], page 23). It runs the `start`, `init`, `toRecord` and `fromRecord` methods on objects and is usually the only object to do so.

5.5 class Server

The Server object just (of which there is only one in each MOZ) sits and watches the MOZ's TCP/Ip port. When it recieves a connection, it instantiates a new object of class Connection and calls the `start` method on that object. It also instantiates an object of class Parser. It then connects both the Connection and Parser objects to the Player object so that the player's commands can be processed.

5.6 class Described

The Described class is not usually instantiated. It adds name- and description-related capabilities to MozBase. It has no location or contents attributes.

5.7 class Located

The Located class expands upon the Described class to add location information, and read-only operations upon that information.

5.8 class Mobile

The Mobile class expands upon the Located class to add write operations on location information.

5.9 class Location

The Location class expands upon the Located class to add contents information, and operations upon that information.

5.10 class Player

The player object holds all the attributes, methods and verbs associated with being a player of the MOZ. The Parser object associated with each player is given full capabilities on the Player object in question.

5.11 class Wizard

A Wizard is just a user with more extensive capabilities. The Wizard class provides commands to use those capabilities more effectively, although being of the Wizard class does not give one Wizardly powers, nor is the converse true.

5.12 class Puppet

The Puppet class expands the Player class into a form suitable for use as an automaton, and includes some basic ability to respond to its master's commands.

5.13 class Container

The Container class is a merging of the Mobile and Location classes, i.e. a Mobile with contents.

5.14 class Exit

The Exit class connects two locations. Each Exit is a one-way pointer to the `destination`. An Exit object in a room will respond to verbs such as 'go' and its own name, and put players through to the destination location.

Since it is both an important operation and one that exemplifies a lot of issues, here's what happens when the player wants to move from one location to another:

1. The player's command is parsed and, eventually, a 'go' method (or whatever) is called on the exit.
2. The exit asks the current location to give the player to the destination location. The process goes something like this:
    1. Updates the player object's location information with a given argument.
    2. Gives capabilities on the player to the remote location.
    3. Removes the player from itself.
3. The exit decides if it wants to allow the operation. This may include examining the player's inventory for a key object of some kind.
4. Note, however, that the exit itself does not have the capabilities on the player required to do the movement; those belong to the location itself.

5.15 class Gate

The Gate class expands on the Exit class with the ability to use an Oz ticket to connect to a room in another MOZ. It gets this ticket from a Terminus class object.

A Gate must be blessed by a Wizard to be able to connect using tickets.

5.16 class Terminus

A Terminus class object is just a Location with the added ability to generate an Oz ticket for itself, to be used by a Gate.

A Terminus must be blessed by a Wizard to be able to produce tickets.

5.17 class Connection

Objects of class Connection have no Active Object wrapping, as they are created on the fly be the server.

They accept input solely from a TCP/IP port which they parse *very lightly* and pass to the parser object. By 'parse' I mean, basically, break into lines. They also accept streams from the Server object, which the Server treats as output strings and passes back out the TCP/IP port.

5.18 class Parser

Objects of class Parser have no Active Object wrapping, as they are created on the fly by the server.

There is one Parser per player connection. It accepts input from the Connection object, parses it, and goes looking for verbs that match it, which are then called with the parsed arguments.

There are, however, situations in which entering new commands should not be possible. In particular, situations in which user input is required to complete a command, such as a yes/no dialog. For these occurences, a procedure on the Parser object is passed down the call chain. This procedure can be used to suck additional input. It uses a simple capability which is revoked as soon as the procedure returns back to the parser.

5.19 class Control

This class is used to create an ohat can be used to control another object. Every time a player creates an object, another object of class Control is also created Storage, and is given the ability to retrieve perfect capabilities for the object the player asked for. This control object is then placed in the player's inventory, while the object the player asked for is left unlocated.

A second copy of the control rod is placed in Limbo, for Wizardly use.

Please note that Control is used to perform mutations, and as such cannot itself be mutated. This is quite deliberate: mutating a Control class object could very easily give access to specialized capabilities.

5.20 class ClassControl

This class is used to write to and recompile classes created by users. It is created when the player runs a "create class" command.

Like the Control class, objects of the ClassControl class cannot, and should not, be mutated.

6 Storage Issues

6.1 Loading And Unloading

The Storage object for the MOZ is the first object to be run. It loads all the classes it can find when its `start` method is run and sets up dictionaries and such so other objects can find each other.

MOZ is disk based, that is, there is no loading of objects at the beginning of the MOZ, besides some of the one-instance objects.

This turns out to be quite trivial in Oz: when objects are loaded, they are pointed to by a weak dictionary. When they are no longer pointed to by anything else, the weak dictionary facility is used to save them to disk.

After the main program loads the Storage object, it loads the Server object and runs that. Everything else is loaded from there as needed.

When an object is loaded, any of its attributes that are of type `objectRef` (as defined in the `exports` list) are instantiated not with the appropriate object wrapper but with a procedure defined by the Storage object that checks if the object is currently actually loaded from disk. If it is, the procedure returns the wrapper. If not, the procedure loads the object from disk and then returns the wrapper if not.

A full-shutdown save process goes as follows (individual object saves can be inferred): for each entry in the dictionary that maps objects to file names, call the object's `toRecord` and save the result of that to the appropriately named file.

Objects are stored in files named <num>.obj, where <num> is a number that is only used internally. These files are saved in direcories named after the class of the object.

6.2 Associations

Objects and classes are stored in files, and must be stably accessible between invocations of moz. Therefore objects and classes must have static names which are used to refer to them by other objects within the MOZ, and there must be associations between those names and the objects themselves (well, the active object wrappers really, see Section 4.5 [The Active Object Wrapper], page 7), as well as between the file names the objects are stored in and their MOz names (or the objects themselves). All these mappings should be two-way, but mappings cannot be made on procedures or objects or classes, so in some cases large searches must be performed. Oh well.

Since class Storage is in the best position to obtain this information, and it has nothing else to do after startup anyways except swap objects to and from disk and answer capability questions, associations will be stored on the MOZs Storage object.

As can be seen in the MozBase def, all objects have a stateless NewName associated with them, which is set at object creation using a name created by Storage, and passed to `init` on the object. class Storage has a dictionary to associate this name with the actual objects: 'oznameObj'.

Since there's no real point worrying about making up file names for objects, and since I'd like to not require uniqueness of object 'names' (distinct from the Oz-level name for the object), I decided to just have the object files be named with incremental integers with

a .obj suffix. A dictionary 'fileOzName' associates these integers with the unique object names. It would have been an array, but arrays in Oz can't grow without rather a lot of work. A dictionary 'oznameFile' provides the reverse mapping.

Objects are stored as records (using `toRecord` and `fromRecord`). One of the things that is in those lists is the class identifier, which we might as well have be a name. That name must be unique. The file name the classes are stored in is just that name with a .class suffix. The file does _not_ store a pickle of the class: it stores the actual source of the class. Otherwise it would be very difficult to present users with the source of classes, since Oz does not appear to have a decompilation feature.

The association between class names and the classes themselves are handled by the dictionary 'nameToClass'.

It is useful in the extreme to store on Storage what class each object belongs to, so that we can certify objects as being instantiations of a known-safe class. So, at load time and during upgrades the class of any given object is store in the dictionary 'oznameToClassName'.

Also, for wizardly purposes it is necessary for the Storage object to store a list of the capabilities available on each object in the MOZ. This are updated at load and upgrade time, and are stored in the dictionary 'oznameToCapabilities'.

It would be incredibly inneficient to actually trace these dictionaries for every call to every method not on the local object, and it would make things much more complicated when dealing with objects from another MOZ See [R1], page 2, so these associations are, for the most part, only used at load and save time. `toRecord`, `fromRecord` and upgrade use these assocations, and that's pretty much it, except for Wizardly usage, which is not discussed here.

7 MOZ Records

There are several home-grown record formats that are important to the MOZ, and they are specified here.

7.1 object records

The object record type is fairly simple:

```
object(
    wrapper: <P/2 Wrapper>
    ozName: <Name>
    capabilities: capabilities(
init: <Name> start: <Name> stop: <Name> ...
    )
)
```

It just holds the active object wrapper, the ozName associated with the object, and the capability list. The only reason for the ozName field is that having that information handy is amazingly Useful for storing the object record to disk.

An example of how to make an object record, from within an object, for giving to another object[1]:

```
object(
    wrapper: self.wrapper
    ozName: @ozName
    capabilities: capabilities(
createObject:
    self.capabilityDict.createObject
objectRefFromRecord:
    self.capabilityDict.objectRefFromRecord
getClass:
    self.capabilityDict.getClass
upgradeObject:
    self.capabilityDict.upgradeObject
    )
)
```

7.2 verbs records

Verbs records are rather complex:

```
allVerbs(
    en: verbs(
get: verb(
    method: get
    parses: [
exampleParseName(
    leftMostPreposition: lmp(
```

[1] This really belongs in the programmer's manual.

```
prep: [ 'from' 'out of' ]
leftSide: lhs(
    objectName: getObject
)
rightSide: rhs(
    objectName: fromObject
)
    )
]
    )
)
    )
    lb: verbs(
cpacu: verb(
    method: get
    parses: [
exampleParseName(
    objectName: getObject
    objectName: fromObject
)
    ]
)
    )
)
```

The outermost record is rather like the strings record. Each language's sub-record contains one field for each known verb. Each verb record contains the method that verb is associated with and a list of parse records[2], which contains instructions for parsing the user input (less the first word, which is always the verb) and turning it into arguments that the verb method will understand.

The verb record is used as an index on a dictionary which is used to store the parsing we have done and to make sure that we don't parse the same input the same way more than once if we can avoid it.

The example above would be appropriate on an object with a method like this:

```
meth get( getObject: GetObject fromObject: FromObject )
...
end
```

7.3 string records

String records are quite simple:

```
string(
    en: "This is the initial room.  It is very boring."
    lb: "ti pamoi kumfa .i .a'ucu'i to'e cinri"
```

[2] The parse record part of the verb records is in such a drastic state of flux right now that I'm not even going to try to document it. It eventually needs its own section, but that might belong in the programmer's manual.

```
)
```

As you can see, one field for each supported language, with a string in the field. Nothing special.

8 Security In Detail

8.1 Initial Capabilities

Note that this section is quite likely to vary depending on the design of the specific MOZ in question. What is described here is simply from the perspective of the base MOZ server.

In general, a created object should be passed, as an `init` argument, the smallest number of capabilities possible. At a minimum, this consists of the following:

- A list of capabilities on the Storage object, so that new objects can be created. This would include a way to list class names as well as a direct object creation call.
- A class testing capability from the Storage object. This takes an object reference and returns true if and only if the object in question is an instance of one of the built-in or otherwise judged safe Oz classes. This is used to warn players that they might be in trouble if they go into, say, rooms of other classes. Note that Wizards can add to this list, and should only do so if they've read the code in question.
- A listContents capability on the room the object is created in. Note that this returns a list of requestCapability capabilities on the contents, i.e. a set of functions that can be used to ask for other capabilities
- A listExits capability on the room the object is created in.
- A requestCapability capability on the room the object is created in.

That's a pretty short list, but it is actually quite sufficient, as we'll see in the next section. Note that this glosses over some issues, like the capabilities shared between the Player, Parser, and Connection objects, but that's a little low level for this discussion.

Note that the requestCapability capability on the starting room allows the object to get the other capabilities on the starting room that it starts with; they are provided initially simply to save time.

8.2 Getting More Capabilities

Note that this section is quite likely to vary depending on the design on the specific MOZ in question. This is simply from the perspective of the base MOZ server.

So, you're a newly created player in a MOZ, with the capablities listed in the previous section. You (or rather, the code underlying the actions) want to say something. Here are the steps involved:

- Call listContents on the current room. This returns a list of requestCapability capabilities for all objects in the current room.
- Call each of those capabilities, requesting a 'tell' capability. Some of them will refuse. That's fine; if they don't want to talk to you that's their business.
- Run the tell capability on each of them with what it is that you have to say.

Note that talking is _not_ handled by the room, in contrast to most MUDs. This is because in a capability based system, we want to avoid testing as much as possible, the testing in this case being whether the object wants to hear your tells or not. This way you can cache the capabilities for all the objects in the room you're in, so the test only gets run

once (when you request the capability). Note, again, that certain things are glossed over, like retries for failed cached tell calls: the object may have revoked that capability, but may not mind giving it to you again.

Note further that although there are capabilities given to you for every object in the room, this gives you no information other than that an object is there; not even the name of the object or any description information can be given to you if the object does not allow it (unless the room has been coded to 'cheat' and store capabilities it is passed and give them away freely). This is how 'invisible' objects can be implemented, although you can still inform the user that an 'invisible' object seems to be in the room.

Another example: you want to move to a roomed joined to this one by an exit.

- Call the listExits capability, use the capabilities provided to request name capabilities, check the names, figure out which exit you want.
- Request the use capability on the exit. If you don't get it, well, you can't go that way.
- Call the use capability, pass it a capability on yourself allowing updating of your location information (i.e. the capablities on the current room). Note that the updater in question should test whatever it gets passed, to make sure the capabilities it was given actually work!
- The exit should already have an update capability on the room it's attached to, which it uses to remove you from the room's contents list, and add you to contents list on the other side. It then calls your update capability with the information from the new room. Any error causes everything to be changed back.
- Revoke the update capability you passed the exit, stopping further updates. In fact, the update capability shoud do this itself.

There is one other, rather different, way to get new capabilities: creating a new object returns a list of all available capabilites on that object.

8.3 Capability Implementation

Capabilities are normally going to be stored in records on various attributes. The format is going to be: capabilities(foo:<ozname> bar:<ozname>), where foo and bar are methods on the object. Theses lists will be seperate from the attribute holding the reference to the object wrapper procedure. So, for example, the list of exits from the current location would be retrieved as follows:

```
{@location.wrapper @locationCapabilities.listExits listExits($)}
```

Note that this means there are two attrs for every capability list. When that request for listExits comes in, the object wrapper selects the appropriate method from its dictionary which is keyed on the Oz names of the methods, and calls that method.

This way of doing things unfortunately breaks the abstraction of treating the object wrapper as though it were just another object. It was possible to make it much cleaner, but it made saving objects a gigantic pain and added a bunch of code. There were other ways of handling it without those drawbacks, but they all put a fair bit of knowledge burden on the coder in terms of understanding how Oz constructs records and such.

If you really want to make a function that returns a function that acts just like a normal object (which I don't particularily recommend, by the way: it'll probably slow things down a bit), here's what one would look like:

```
proc {CapProcMaker ObjProc CapList ?CapProc}
    proc {CapProc Meth}
{ObjProc CapList.{Label Meth} Meth}
    end
end
```

Note that it is very important that you not try to save this procedure on your object (i.e. do not add it to the exports variable of the class), as this will cause the saving to fail. This means that if you are storing this procedure on an attr you need to check for its existence (probably using IsDet or Value.type) pretty much every time you use it, or make sure it gets recreated every time your object is reloaded (i.e., create it in your `start` method).

8.4 Persistence

Persistence is something that is entirely normal in MUDs, so this whole section may seem amazingly obvious to those of you used to MUD programming. However, it is very *abnormal* in operating systems, but see EROS.

In an operating system, if you want to open a file, you go and get a list of files, then ask for permission to open it. This makes it easy to deal with reboots: you just go look for the file again.

Obviously, this can't work in a capability system: where do you get the list of files from? In the context of a MUD, there would need to be some way to search through all the rooms to find the one you were in to ask for capabilities on that room. There is no such ability. OK, that's not quite true, the Storage object can, and does, have such an ability, but *only* for purposes of implementing persistence.

So, the mud simply stores everything. Every few minutes, each object is written to disk, along with all the capabilities it has acquired. So there is no rebooting, really: after a crash or whatever, the MOZ is brought back up as though nothing had changed.

Actually, that may not be quite true: only the capabilities that are saved as part of running `toRecord` on that class are stored, so stuff that is only being stored in running code or cache attrs is not preserved.

Capabilities, because they are simply implemented as a record of Oz names, are stored as per normal by the Storage object. The object wrapper handles are turned into Oz names on the way out and converted back on the way in.

For that to work, the newly loaded object that the capability is for must remeber what the names were that it had associated with various methods, especially since this is handled in the wrapper. The way that this is done is to simply have the information be stored as a dictionary in the wrapper; the wrapper has code in it to intercept the `toRecord` call and add the capability information, and also to intercept the `fromRecord` call and extract it again.

8.5 Wizardry

It would be possible to have a capability based system with no user having more power than any other user, in other words no Wizards, no Gods, no root. Of course, because of the nature of capabilities there's nothing to stop a user from setting up their own little domain in which they have ultimate power, but that's not the same.

However, as a design decision I feel that that's a bad idea. Capabilities, as previously stated, make it rather easy for users to screw up if they try their hand at programming, or go into an area owned by a malicious user and say 'yes' to a lot of request messages and end up in a really bad way. I would like there to be a way for them to get un-screwed.

Therefore, MOZ supports Wizardry by default. Being a Wizard is very simple: you have full capabilities on the Storage object, always has the most up-to-date set of capabilities for every object in the MOZ.

Note that as the wrapper code is just about the only ex-DB code, it is impossible to circumvent the fact that the wrapper code will report capability changes to the Storage object.

Note that users can get around this by creating objects which they do not report to the server. This has the substantial disadvantage that they will be lost upon server restart, as normal users are not given capabilities to anything that would allow them to store the information.

Also, the wrapper tells Storage to update this information on each change.

9 Unsorted

- As a stylistic issue, all methods should use full record field names for their arguments.
- I am currently of the belief that all special user capabilities should be stored on objects (i.e. keys and such), primarily because any other method is just too damned easy to spoof. Here's some stuff about another way of doing it.
  - It is useful to have capablities on objects that are, more or less, automatically activated (i.e. a door that only you can walk through). MOZ calls such capabilities 'keys'.
  - Keys are stored on their own attribute on the user.
  - When a verb is call, a requestKey method is passed. Actually passing a key requires asking the user first.
  - Keys are in a dictionary indexed by ozName; ideally there would be some way to verify the ozName of the calling object.
- However, having special capabilites on objects (which are presumably carried) leads to some issues with things like a dragon who doesn't eat anyone wearing the Gauntlet Of Might, or whatever. Ways of handling this are below. I have not picked yet, although I have a bit of a preference for a 'wear' verb.
  - Verb transparency to objects in inventory, which could cause problems if you pick up the wrong thing. But this could be selectable, i.e. some objects are allowed to be verb-trasparent. The easiest way would be a 'wear' command, such that objects that are word are checked for verbs.
  - Extensible verb lists (i.e. an attr?) in which an object field is part of the verb definition. This way the Gauntlet of Might would add a verb to the *player* (with permission, of course) called 'challengedBy'. Calling challengedBy(arg1: <the dragon in question>) returns a value that causes the dragon to calm down.
- This was in the class Parser section. I was smoking crack.

  > Note that the verb call can immediately thread off, and should almost always do so if any non-trivial computation is done, since output is handled by an Oz port. This will be particularily important when in-game thread management is implemented; threading off will allow the player to kill threads if necessary, as commands can still get through.

  There is no way to get a list of threads in Oz, so what needs to be done at some point is to hack the parser object to thread off all verbs and save their thread ids on the player object, so that the player can kill them.
- I have no idea where to put this. Movement of an object is three seperate things, in terms of capabilities: changing of where the object's location attr points, and changing the contents list of the before *and* after objects. Therefore, capabilities on three objects are involved. This should be discussed, both as a capability example, and an implementation detail. Maybe it's time for implementation notes?