Mining MUSE (A Moderately Useful Source Exploration)

This is a different kind of project for Computercraft, one aimed at teaching, by example, what's involved in developing maintainable code. It's a Moderately Useful Source Exploration, MUSE.

MUSE is a collection of resources built on the ComputerCraft *mod* to Minecraft written in Lua . It provides a set of capabilities for ComputerCraft turtles and computers (also written in Lua) that make it easy (quite possibly too easy) to mine, farm, and do explorations in Minecraft worlds.

Looking about in MUSE, as we'll do here, is a different kind of exploration. It is the exploration into how to develop clean, maintainable code intended for those worlds (as a start). Making a mess is easy (quite possibly too easy). The hope that what we're about here is makes it easier to avoid.

But why would you want to do that? Because, even if you're working all on your own, looking at what you (and your trusty AI agent) wrought in fevered moments not long past, you just know deep down that it was some rogue AI from another planet that wrote the tangled mess you're facing. And, of course, in the real world, code development is generally something done with others, sometimes a lot of others (and their agents). It's good to play well with those others.

Can you just let the agent do it? Of course you can. But then you're still responsible for that possibly hallucinated, and maybe hallucinogenic mess. Better you know what the code looks like that you'll need to maintain.

And there's a deeper reason. It has to do with the overall design of what you're responsible for. There are just a lot of broader design choices that are reflected in the code but got decided way before the code was written (whether generated by you or your agent). We'll get to those shortly.

Chapter 1 - Exploring Why, What, What Not, and How

Mining MUSE is based on the idea of exploration. Just as software development is actually practiced, there's a lot of exploration to do when trying to understand and fix or extend someone else's code (including the mess made by that AI cowboy). What follows in these chapters is an overview of that exploration. But the overview is only the root of a very branchy tree. The interesting stuff is generally found somewhere near the leaves. Where exactly depends on you, your interests, your motivation, your experience. Go as far as seems useful. Go back and look further when that seems right. It's up to you.

As you explore, after some looking at some fundamental ideas for maintainable code and the design of a code base, you can use MUSE to look at some broader design choices:

- choosing the right abstractions for your design,
- dealing cleanly with the inevitable changes to the user interface,
- considering the issues with persistence over time,
- making available and managing concurrency,
- handling errors on a computer in a network far, far away,
- providing distributed service discovery,
- establishing remote procedure calls and remote command interfaces,
- looking at the role of 'little languages" declarative programming, and finally,
- deciding whether frameworks (an inversion of control) would be helpful (or not).

The (HTML) links to places in the branchy tree throughout MUSE are a way into the exploration. The materials in the tree describe working examples with, yes, more links, drawings, pictures, and

extensively documented code. There's probably as much explanatory text as executable code in the examples. This leads to the usual tension between the ideas of *literate programming* and the, all too correct, understanding that "comments (sometimes) lie". If you need to extensively comment code, as we do to meet our goals here, you're taking on an extra burden to keep the comments true to the code (which never lies). There is a benefit though of doing the work of extensive commenting: you're explaining the code (especially the *whys* of the code) to a virtual partner when a real life one, quite possibly the you from another planet, is not available. Often enough while explaining, you'll see something you might otherwise have missed.

Explore the links in the code body to go on excursions further (and further) afield. Feel free to wander, skimming as you go, and then return for more extensive explorations as seems best. Early chapters will generally go into detail omitted in later ones so if you skip chapters, you may need to go back a bit to go forward.

So then, here we are, looking at whether to knock about in that tree. Is it worth the time? For that, let's examine what we're trying to do. Here it is:

- **Why this?** The intent is to provide an exploration of how to write clean code expected to be read (so it can more easily be maintained) and, as as part of that maintenance, evolved and extended.
- **What is this?** A tutorial of sorts using just the code needed to implement a particular set of ComputerCraft capabilities with Lua scripting. Ease of code maintenance over the long term is the chief goal of the design of that code. MUSE is a bit strange in that it aims to be a

professional piece of development for a problem domain which is, after all, just a game. The game is the bit of sugar that, hopefully, helps the medicine go down.

- **What is this not?** It does not pretend to be a comprehensive review of software fundamentals , computer programming or, the range of ideas for the design and use of computer languages. (Here's a tour guide you might find useful.) It's also not likely to be useful as someone's very first exposure to just getting some code to run. There are videos and other on-line resources as well. (But do come back once that's behind you.) A more serious limitation is the issue of scale. MUSE is not millions of lines of code created by scores of developers. The hope is that by exploring MUSE you will find ideas and approaches that will prove useful if you ever need to work in that necessarily more constrained environment.
- **How is this done?** By exposing in some detail the design ideas underlying a particular body of code, namely the code for MUSE.
- **What's next?** Reading on to review the design and implementation of MUSE.
- **And after that?** Setting up a environment allowing modification of the code base and trying it out in ComputerCraft execution. That's described in an *appendix*.

Chapter 2 - Moving Turtles: Functions, State, and History

One of the first things to think about when designing a body of code is how that body of code might be organized. MUSE is organized around the idea of what we'll call a *module*. Each module is a file that includes at least one *library*. (A module may contain additional libraries sharing some internal implementations.) In what follows we'll explore the idea of libraries, likely the most important idea in the exploration of MUSE, and show an example of how they might be organized for readers and maintainers.

Library interfaces set the boundaries of a library. They indicate what can be relied upon by the users of a library without needing to know about its implementation. In this way, they allow the effects of library maintenance to be contained within a library. The right organization of library structure is an important way of easing maintenance. Your first attempt, done in haste, might not be right. Expect penitential experiences while refactoring code that was written in a hurry. (Code in haste, repent at leisure.) Sometimes, it's best to just step away for a bit and just think about a design (or debug) issue. Don't just do something, stand there. Really. Often a cleaner approach to the problem will emerge. If the cleaner approach solves more than one problem cleanly, it stands a good chance of being more maintainable.

As we've said, libraries have interfaces. MUSE introduces a way of describing interfaces and checking their use by adding annotations as comments to the source code. As a dynamically typed language, Lua provides flexible and expressive richness for implementation and the corresponding opportunities for run time errors that might only show up (expensively) in fielded code. Describing interfaces and checking their use is a way to deal with some of these unfortunate opportunities as part of development. We do this with CodeMark Annotations documented in their own right. It might be best just to scan that documentation for now to get a general sense and come back to it as needed. The descriptive text for the code we'll explore will be enough to understand each interface (although in a less concise form).

The second idea we want to think about as we begin our exploration deals with the management of *state* . The most common ways to design code involve the concept of state, elements with values that are changed (*mutated*) during code execution. Mismanagement of state, especially inappropriate exposure of state, is often the root problem for a lot of code that is difficult to maintain. A radical approach to the problem of state is to do without it. There are purely functional languages like Haskell that go there. Lua doesn't.

The third idea we'll explore is some of the ways Lua functions can be used to structure code cleanly to manage state and more. This idea is not at all limited to Lua. Lua is just one example of a language where functions play such an important part in code structure to do this (and a lot more).

There's a lot of design power in an approach that exploits functions as just another type in a language. There's a lot of maintainance power in basing a design on tables of functions and other controls to direct the work done as requirements evolve. There's also a lot of unavoidable complexity in dealing with error conditions that crop up even in the relatively friendly environment of a game. All this is to say (as we've said) that the implementations we'll look at are not what you'd expect as the very first exposure to just getting some code to run.

The two libraries we'll explore in this chapter are foundational in two ways. As you might expect, they expose function interfaces that many other libraries depend on. But just as foundational, they define common data structures, the turtle's `situation` and the (named) `place` in the Minecraft world where a turtle (or the player) might be found. Many MUSE libraries work with these structures

With that understanding, it's time to climb about in that tree we spoke of. Try clicking the following link to explore the implementation of the `lib/motion` module. Then look at `lib/places`, its companion. Explore the structure of the libraries in these modules, the issues they raise in dealing cleanly with state, and some of the ways to use Lua functions to structure code. The exported interfaces for `lib/motion` and `lib/places` might help you to see where you've been climbing. (Just to start out, it might be better to just scan summaries for now. They'll mean more to you after spending some time on implementations.)

These libraries rely on `lib/core`, a collection, as you might expect, of generally useful (core) functions. For now, just look at the exported interfaces for that library. We'll discuss the implementations of these functions as needed.

As you finish your exploration of `lib/motion`, take some time to consider how to test it. For MUSE, the *unit tests* for MUSE libraries are incorporated in the `tests` directory. (In some practices, such unit tests are expected to be written before coding. You might want to try that.)

MUSE includes rudimentary support for regression testing for sets of unit tests in `lib/check`.

When you're ready, let's press on to the next chapter where there'll be ideas about what to expose and what to hide in a design and how to build a simple and easily maintainable command line user interface.

Chapter 3 - Choosing Abstractions, Tables Making Functions, Commands Thin

Information hiding is an important part of what a library does. The choice of what to expose as an interface is the choice of what to hide. One aspect of choosing what (detail) to hide is choosing the abstractions to create. Exposing the wrong detail gets in the way of delivering an evolving yet maintainable system. Thus, the task of design is often about exposing the right abstractions.

For MUSE, some abstractions are obvious. They closely relate to what's in the game, our *problem domain*, and have fairly intuitive expression in the *solution domain* that we're building. For changes in the position and orientation of a turtle, the right abstractions might expose interfaces for movement in the four cardinal directions ("north", "south", "east", and "west") rather than relative directions ("right" and "left"). The `lib/motion` library exposed these abstractions for movement.

The `lib/turtle` library extends this abstraction consistently with the other interesting things turtles can be made to do. The consistency itself helps in maintenance. As we said, ComputerCraft's primitives deal with turns "right" and "left". They deal separately with those other things turtles can do in terms of "forward", "up", and "down". A useful set of abstractions would be to hide all this detail and expose turtle interfaces uniformly in terms of six directions: the four cardinal and the two vertical directions (while still allowing `forward`).

There's a pretty long list of turtle operations that would need fitting into this abstraction. The fitting is so repetitive that it lends itself to representation as a table. There's a couple of maker functions executed when `lib/turtle` is loaded. They make the functions that `lib/turtle` exports. Each of those exported turtle operation functions is then exposed as a table indexed by `direction`.

Functions making tables is the *dog bites man* story (it's not exactly news that *functions make tables*). With a little license, this is the *tables make functions* story. It's a small step toward a declarative programming style. Larger ones to follow. For now, look at the `lib/turtle` implementation to see how to have yet more fun with functions.

Another detail needing abstracting is the notion of what's in which `slot` of a turtle's inventory. What's where exactly is something worth hiding. What we generally just want to know is whether the turtle's `inventory` includes certain items. Or better, whether it includes certain kinds, `categories`, of items. These might be, for example, `fuels`, `stones`, or `ores`. You might want to know about whether a turtle's got stones but not care what kind exactly. A category could be items useful as `fills` or the kinds of `dirt`. We might be more interested in the total fuel energy available to a turtle rather than any specific kind of fuel.

Finally, `lib/turtle` also abstracts dealing with some of the difficulties turtles face when attempting to move to a position. Hiding the detail of dealing with obstacles presents a simpler notion of movement.

With all these changes, `lib/turtle` has superseded the ComputerCraft definition of a turtle. Behold, all turtles are made new. (At least if you load `lib/turtle`.) There are two clients of the born again turtle that we'll discuss next: `lib/roam` and `lib/task`. First `lib/roam`.

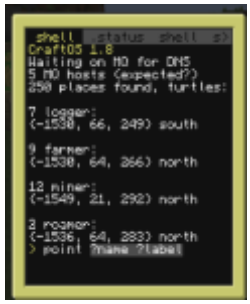
The `lib/roam` library provides yet more movement extensions for our turtles:

- ***try again strategies*** controlled by a `.start` parameter to look for ways around blockages,

- ***going to a `place`*** (as well as specified xyz coordinates)
- ***turtle side support*** for following the `player` around (as befits, say, a squire turtle), and
- ***a very tiny language*** to chain together commands to move a turtle piecewise in all those abstracted directions,

All this is done using both the abstractions of `lib/turtle` and the primitives of `lib/motion`. The library also shows a way to design a more maintainable *Command Line Interface* (CLI) based user control facility. The `lib/roam` library centralizes error handling, safety checks on turtle dead reckoning, and simulated blockage testing for all the commands it supports. It does all this in its CLI dispatch mechanism, one part of keeping command code thin, a virtue eagerly to be sought.

Managing the inevitable and ongoing changes to the user interface is an important issue for maintainable code. MUSE keeps the actual UI code "thin" in the command programs that provide the UI. All the hard stuff is in the support libraries. Changes to the UI itself are generally pretty easy. That's good because experience with a UI is the most likely generator of reasons to change it.



(We have it a bit easier since the user interface for MUSE is just a CLI. The commands are simple text inputs to the CraftOS command line. The design issues for a graphical user interface, a GUI are the subject of a whole other exploration.)

A desire to ease testing is a second motivation for minimizing the work of the command itself. It's easier to test function interfaces than creating the testing facilities that drive simulated user interactions. Because of this, almost all of the work of providing MUSE CLIs is done by libraries supporting the commands. More elaborate command interfaces (including GUIs) would require more elaborate testing support systems. Sometimes project goals demand such interfaces. They would be distractions for MUSE.

Lua has a one pass compiler. Functions build on each other from the beginning of the file to the end. The tree image links to the beginning of the file. Alternatively, you may want to follow the code backward toward the beginning through the support for `go` and `to` and then look at `come` to see how this is done. Or just read it as written by clicking the tree.

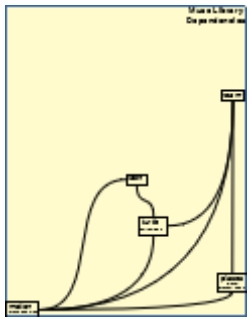
You might have been driven to abstraction by what we've been going on (and on) about. But it's the new and improved turtle that lets us build the `lib/task` library. It supports a CLI for low level turtle tasks in terms of those abstracted directions. Some, like `dig`, involve chewing gum while walking. These use the `step` closures from `lib/motion` to do the work, as for `dig`, one small step (for a turtle) at a time. The implementation for each of those low level turtle operations in `lib/task` uses the table (indexed by `direction`) that was made for that operation by `lib/turtle`. Oh, the things a turtle can do.

There's a technique in the `lib/task` implementation that folds particular kinds of movement vectors into a (scalar) `direction` for that vector. It takes advantage of a property of the sequence of some such vectors that they only make changes in one axis at a time. It's the same sort of thing

as done in `lib/map` to settle on a compass direction. You may find use for the technique in this or other kinds of folding.

A similar pattern as described for `lib/roam` is used for the implementation of this CLL (CLI support library). There's little need to see how, in particular, `lib/task` does this unless to see another example of how thin CLI support might be built. But follow the link if you're interested.

If all you're looking for right now is how these libraries fit into the overall design, here are links to the interfaces for `lib/turtle` , `lib/task` , and `lib/roam`. Working on code this way is a good strategy to have available to you. Sometimes in coming to grips with a body of code written by that visitor from another planet, just understanding the library interfaces for what's been exhumed can be helpful.



Additionally, there's a drawing, the one that's thumbnailed here, showing how these MUSE libraries work together with what's gone before. Together they form and make use of (wait for it) increasing levels of abstraction for MUSE.

When that's done come back here to explore how to establish state that persists across *sessions*, how to deal with state distributed over a network, and how error handling works in network environments.

Chapter 4 - Persistence, Concurrency, Remote Errors

This chapter is less about how to write code in general. We've pretty much finished flogging that particular horse. This chapter (and pretty much all that follow) is more about how to approach a design and how to provide some important mechanisms dealing with networked computers. They're the ones that you may need to design (or understand how someone else did).

MUSE needs to deal with networked computers (like turtles and pocket computers). They'll be operating on distributed persistent state. The operations may throw exceptions to signal errors. We'll discuss in a bit how to deal with them. But first, persistent state itself.

The persistent state is stored in *non-volatile memory* represented in ComputerCraft as disk storage local to each networked computer. Most commonly, such persistence requires that the mutable (changeable) elements in computer memory be *serialized*, that is turned into *strings* that can later be deserialized in order to store the serialized elements back into computer memory to do operations on them there.

Supporting *networked* distributed state is done using these same serialization facilities to send strings over the network. These strings are then deserialized when they are received by the target computer. All this (and much more) is done by the MUSE `lib/core` library .

As promised, this chapter also deals with remote error handling. Exceptions thrown by remote operations on remote computers to signal errors need to be caught on those computers and dealt with so that the remote computers remain available to handle future commands. Error information then needs to be passed back over the network to let the remote caller know what blew up and figure out what to do about it. Check out the `lib/map` dispatch and the `lib/core` handler to see

how this works. Then follow a couple of the links provided there to look at the actual command programs. It's another, more developed, example of keeping the CLI thin. We'll see later how the same dispatch is used whether a command is invoked locally by the player for the player or the command arrives over the network for a turtle.

Finally this chapter deals with the issues raised by concurrency. In-game, all of the simulated computers including those for turtles and players are, of course, supported by one real computer running the game. This is mostly transparent to MUSE except for the need to yield control from one of these simulated computers to the real one from time to time so other simulated computers get their turn. Turtles and other in-game computers then need to re-acquire control to handle what's been happening to them.

Lua provides and ComputerCraft uses *coroutines* for *cooperative multitasking*. MUSE programs yield control implicitly whenever there is a turtle movement or when waiting for a network event. They yield control explicitly through ComputerCraft `os.sleep` calls. Yielding control either implicitly or explicitly allows, for example, network operations and GPS calculations by other simulated computers to proceed. MUSE handles network operation events using ComputerCraft's `parallel.waitForAny` function. More elaborate use of coroutines might have been considered. MUSE keeps it simple.

Cooperative multitasking is a simple model that works as long as the multitasking is actually cooperative. Control is relinquished to allow other operations to proceed in a timely way but only when it's not a problem to do so, that is as long as *data consistency* is preserved.

Here's an data consistency example: a network message is received, data structures de-serialized, associated changes made in other data structures, and then, perhaps, serialized and stored. All, really *completely all*, of this needs to accomplished by a cooperating (simulated) computer when it has control or none of this. If it does any of this, it needs to do it all before starting anything else. Doing just part leaves inconsistent state laying around to make trouble at some later time when it's really hard to figure out what went wrong.

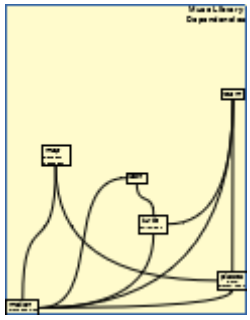
In MUSE there is no shared mutable state between computers. It takes a network operation (applied all or none as above) to change state in another simulated computer. There's no need for the complexity of *critical sections* and the associated opportunities for error. There are, indeed, well developed *language models for concurrency* to handle concurrency in a maintainable way. But if you can meet your design goals, your work will be simpler and more maintainable if you restrict the need to go there.

In MUSE, all the simulated computers do share a ROM image but this is, by definition, immutable, read only within a session. ComputerCraft makes no guarantees about what happens if there's an out-of-game change of ROM during a session. (There are other considerations for the changes in interfaces that deal with networked persistent data structures. We'll discuss these later.)

If there's one design idea for maintainable code in all the above, it's to seek to keep it simple. Design goals for a project may not allow the simple approach just described for run time organization. There might be tools to mitigate risks inherent in more complicated organizations.

They might work. But perhaps you can find a way to isolate complexity in the design so that most of what is developed can rely on simple assumptions about what happens at run time.

A design handling persistence, management of distributed state, and remote error handling is demonstrated by the implementation and interface of the `lib/map` library.



The library is built on the `lib/motion` and `lib/places` libraries we've already looked at. Here's the interface for `lib/motion`, the interface for `lib/places`, and a drawing showing how these libraries form the foundation for MUSE. We'll add more libraries to this drawing as we continue our explorations.

There's an operation in `lib/map` that needs noting. It's there for a reason we'll get to downstream. However it needs saying now that the `chart` operation is one of those things your parents might have warned you about. (I suppose it depends on the parents.) It's almost a cartoon for a big, gaping, really scary security hole. Invoking it loads and executes a file in the `charts` directory. Which could do anything that a ComputerCraft computer could do.

Now, to be sure, here be no nuclear reactors. More to the point, as someone creating the code for these computers, you've already been empowered to do anything with a ComputerCraft computer that it can do. Nonetheless, if you're tempted to do what I did and do it in the real world, rather than what I said you shouldn't, think again. It might be appropriate. Or not.

With parental warnings duly noted (and forgotten), we'll see in the next chapter, how remote CLIs are provided through a *remote procedure call* (RPC) that uses the local CLI dispatch we've just been exploring to command remote computers over the network.

Chapter 5 - Distributed Service Discovery, Remote Procedure Calls and CLIs

This chapter looks at systems with distributed operations on distributed state. The principle mechanisms considered are distributed name servers to discover providers of available services and remote procedure calls (RPCs) to make use of those services.

An RPC in a *client-server model* lets (ahem) clients get services from, well, servers. For MUSE, the client is generally the player's pocket computer. The servers are turtles perhaps with specific roles for farming, mining, or logging. The roles are established by ComputerCraft computer labels in an area: `farmer`, `miner`, and `logger`.

The `rover` is a general purpose worker acting as something like the squire of the `player` going wherever the player goes. The other turtles, specialized `landed` workers, stay in the place they work, a `site`. The `player` may establish other sites with other sets of landed workers. The landed worker turtles are equipped with tools appropriate to their specific role.

RPCs are directed to turtles as specified by their role. ComputerCraft, of course, has no notion of roles: all it knows are computer IDs (numbers) and computer labels (strings). Its networking is tied to numbers, sequentially assigned as computers are brought into the game. If MUSE operations

were based on these numbers, they would be subject to a dependency that's unwieldy to manage and therefore fragile in practice. (Stuff would break.) We need names, not numbers, for the computers in our network that have roles.

As we've said, roles are tied to labels. What's left for us is to find a way to discover which names as

roles are associated with which numbers. This could be done by a task done whenever a player brings a new computer into the game. There could be a table associating names and numbers updated by such a task and kept consistent (and persistent) for all computers. We've already seen with `lib/places` how keeping names for new things (like `places`) consistent and persistent throughout the network might be done. Systems have been built this way. MUSE assumes a single player. Nothing happens to turtles "in the meantime" without that player having directed it to happen so, strictly speaking, this approach, based on consistent and persistent mappings could work. (Turtles are really hard for hostile mobs to destroy.)

Because it's useful to consider, MUSE implements a a different approach that does not count on managing distributed persistent mappings. Associations between names (roles at a site) and numbers (computer IDs) are discovered by a distributed discovery service, `DDS`, during system startup to reflect the state of the system however it's been changed.

Here's how it works. Roles and sites are established when Computercraft starts up a simulation. When that happens, `.start` , run at startup, executes `dds.hosts`. This gets the computer `ID` for each sited role (saved by Computercraft as a computer `label`) and sends the pair to each computer that has run `dds.hosts`.

At some earlier time, MUSE turtles and the player's pocket computer were assigned roles at their site (as computer labels). This is done by running the `join` command which invokes `dds.join`. The current site is established by the site command implemented by `map.site`.

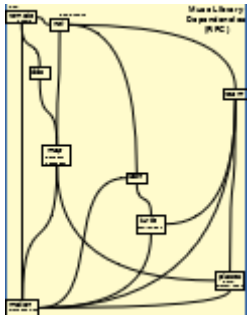
We need to note that name servers come in different flavors with different capabilities. The MUSE distributed discovery service, `DDS`, is only a distant cousin of one real world system, the Domain Name System (DNS). DNS also basically associates names with numbers. But it deals with millions of computers managed with the barest minimum of coordination over the entire world. Our little DDS does not begin to consider such scale or distribution of control. However it does model approaches to the sort of service discovery provided for small scale home or local office networks.

Given a way to associate names wth numbers with `DDS`, we can provide a way to get a service request from a client to a server. The other part of what is needed is a way to actually provide the service. That's a job for Remote Procedure Call, an action hero you may (or may not) have heard of.

The design of the MUSE RPC makes use of MUSE serialization and Lua's `load` facilities to package and unpackage procedure arguments for network transmission and reception. As a development exercise, this could be generalized to use language independent formatting such as JSON . There are a number of such implementations.

But that seemed to be a distraction from our goals. Even so, supporting our goals, the design is more general than is strictly needed. As we'll see, it is built to handle environments that could be bothered by unfriendly, rogue actors. This is not MUSE but it seemed useful to explore.

It is also designed to be testable outside its intended deployment. This kind of design certainly *is* relevant to MUSE and might be important in more than just the MUSE environment. In addition to the debugging benefit, working in a development sandbox means that getting errors when trying stuff has limited consequences.



The `lib/remote`, `lib/dds`, and `lib/net` libraries provide the interfaces for the facilities we've been discussing. The drawing shows how they fit in the overall context of MUSE.

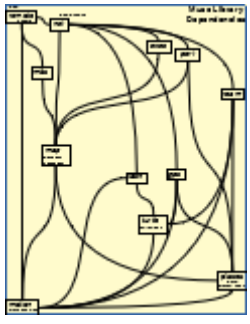
Look first at `lib/remote` to see how RPCs work. Review of its code will lead into excursions of `lib/dds` and more. Then look at how `lib/net` actually creates (and documents) the remote CLI we said this chapter would discuss. The generated documentation for the remote CLI is here. The information in libraries, kept close to the implementation, was all that was needed to generate it. Note that `lib/net` calls on the same dispatch operations in the targeted libraries as used by local commands.

Most of the remote CLI commands are prefaced by a role (`farmer`, `miner`, `logger`, or `rover`) to target a given turtle. A CLI command prefaced by a role needs a dispatch to the `lib/remote` handler for that command. As you've likely come to expect, there's not much to these. They're boringly (and helpfully) similar: `farmer`, `miner`, `logger`, `roamer`.

As a convenience some commands assume a role (and so, a target). For the `rover`, these are `come` and `tail` . You may have looked at turtle part of their implementations in `lib/roam`. The other part of their implementations is in `lib/remote`.

There's a role we haven't mentioned till now, the `porter`. It's associated with a very lonely command computer, just the one of them in the whole MUSE environment. While, for generality, it's possible to send the `porter` a CLI command prefaced by its role just like the others, actually only a very few of them are useful. It's not a turtle.

A library, `lib/exec` uses the `porter` to provide support for setting up MUSE operations. The `lib/exec` library helps aligning Minecraft coordinates with turtle dead reckoning for ComputerCraft GPS deployment with `lib/gps` . This is done by the `locate` command, implemented by `exec.locate`. The library also supports supplying a named MUSE `range` for the Minecraft's `/forceload add` operation. The command is `activate` implemented by `exec.activate`.



Since we've gone ahead and used a command computer, we've already demonstrated a shameful lack of constraint. Another command computer indulgence, the `lib/port` library, provides an integration with `map.places` and the Minecraft `teleport` command. The drawing shows the library in the overall context of MUSE.

There's a bit of application design involved in adding a `teleport` facility to MUSE. The design approach was to make (rough) parallels with real world trips. There's advance booking a trip from somewhere to somewhere else. That generally involves figuring out what the trip costs. Doing the trip with baggage increases the costs. When all that's

settled, actually taking the trip makes use of the booking and incurs the cost. Look at `lib/port` to see how that's all worked out in a Minecraft context.

Astonishingly it's taken us till now to build up MUSE capability to the point where it would be useful to actually explore operating MUSE in the Minecraft/Computercraft environment. If it seems good for you to do that now, here's a link to guide you through the needed `setup`.

In the following chapter we'll look at the MUSE libraries that create and make use of what you might think of as a (very simple) declarative language . They allow us to describe turtle operations by *what* is to be done rather than *how* it is to be done. In some ways, it's another kind of abstraction with the goal of increased maintainability: hiding the detail of *how* while exposing the main point: *what*.

Chapter 6 - Planners and Workers, Declarative Programming

In this chapter, as threatened, we'll explore the idea of declarative programming . We'll look at using this approach for digging shafts and tunnels to mine ores. The *what* of all this will get pretty detailed. In some sense, that's the point. The luxury of just saying *what* is particularly important when there's a lot of detail (that inevitably will get changed). Apologies in advance: this chapter will be unavoidably detailed in the *what* in order to ground our exploration.

The *what* that is to done for our examples is to dig some holes. In particular, shafts down to a given level and tunnels to mine ore at that level. However, digging is only part of the work. Provisioning a shaft needs, for example, ladders, torches, and storage barrels put just right. For the player's safety, we also need platforms to stand on.

Let's start looking at where we want to end up. So that we stay within blocks that Minecraft has loaded , we want to dig (mostly) down rather than out. Straight down has its own preferably avoidable excitements. So we want to dig in a pattern that snakes back and forth digging

- **down and east** (say), then **down and west** (say) to get to a given (odd numbered) level
- and then dig in a complementary pattern of
- **down and west**, then **down and east** to get down to the next (even numbered) level.

The even-odd pattern repeats for as far down as we want to dig.

Here are some pictures of what that looks like. First the ladder that got us down to the odd level:

Next looking east across the platform at that level to the ladder that goes down to the next (even) level:

The ladder on the left of this picture is the same one we saw above. Next, imagining we've gone down that ladder to the even level, looking back up the ladder we took going down looks like the next picture.

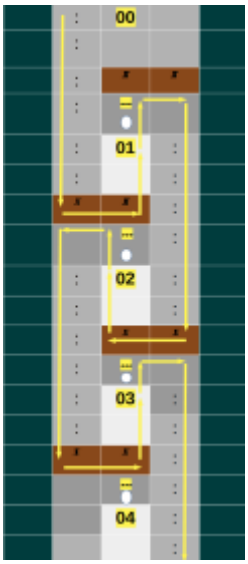
Looking west at that level across the platform looks like the picture below.

As you look across the platform you can see the ladder down to the next (odd numbered) layer. If we went down that ladder to the (odd numbered) level just below, it would look like the first picture in this series. That's good: we have a repeating pattern to code to.

Note the torches placed on blocks of dirt. It's easy for players to get lost in these tunnels. The torches placed on dirt blocks are anomalous in mines. By their oddity, they stand out. Helpfully, they indicate the way back to the shaft along what we've called the *inner* tunnels.

Follow the link to see a schematic representation of what's going on in these pictures. It shows the *what* for the work to be done, the repeating pattern shown schematically. The schematic tells a fairly complicated story:

- how the turtle is to **move up and down** between levels in "the plane of motion",
- where the **ladders and torches** are to be put north of that plane,
- where fill is to be put to form a **platform for the player**, and, finally,
- where the `marker` **and turtle's** `post` is in the shaft at each level.



We'll use (the numbered) markers to navigate the twists and turns of the mine to get to a `post`. A `post` is a place that the mining turtle goes to between operations. Each `marker` is simply a `point` as we described when discussing `lib/places`. It has a particularly structured name and label to make mine navigation work.

All this detail motivates a declarative *what* rather than imperative *how* approach to writing maintainable code. It's likely obvious that getting all this right the first time isn't very likely. There will be changes. It's easier to make those changes if we can find a declarative representation that pretty much directly follows our (changing) schematic as understanding develops of what must be done.

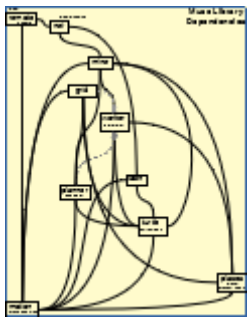
So then, on to how all this works. As is typical, there's two parts to this declarative programming engine: a *planner* (a producer of the *how*, an executable interpretation of the *what*) and a *worker* (consuming that interpretation to actually do the work). Helper libraries provide *work functions* extending the planner's declarative language for specific situations. In MUSE, the *what* is a `plan`. The `lib/planner` library uses Lua to load a `plan` and produce its internal representation. This representation is consumed by `lib/worker`. It's a different kind of dependency. Together these libraries are the procedural engine for the *how* that supports the declarative *what*. They take care of the *how* so you don't have to.

The design of the declarative `plan` "language" is governed by a few principles. We want to:

- **limit the declarative scope** to moving turtles, placing items, and marking places; *and beyond that scope*,
- **extend operations with Lua *work functions*** rather than making a bigger language; *and*
- **fit well with Lua syntax** to minimize the work (and opportunity for error) in parsing the representation.

The general idea is to keep it simple and use Lua for the hard stuff. Making the language bigger than it strictly needs to be means there's more (parsing) to get wrong and, just as importantly, more that's unfamiliar. (That's a potential pitfall for *little languages*.)

One of the strong points for Lua is the coherence between development and operation: Lua ***is*** the *shell language* as well as the general development language. In keeping with this idea, a MUSE plan is just a Lua file. It's easy to escape back to Lua whenever there's the temptation to make the little language a little less little.



As a guide, here's how the libraries we'll be looking at for mining fit together. (Click the picture.) For clarity, the drawing just shows these libraries and their dependencies. As you might figure, there's a lot of factoring (that is, which piece does what) to explore in the design. The `mine` library is a CLL, Command Line Library. It uses`lib/planner` to build plans for commands given it and `lib/worker` to execute them. Work functions for the plan to actually mine ore in our example are provided by `lib/grid`.

Our little language expects to *step* turtle motion in a closure as we discussed way back when we explored `lib/motion`. At each of these steps, the *what* operations specified within the declarative

scope of our little language are executed and then that optional (Lua) *work function* is invoked. The escape to procedural programming keeps the little language little.

Look at the `lib/planner` introduction to see the definition of our little language. Then explore the `plan` for a snake shaped shaft to see how the language is used. This is the plan for providing shafts like the one we've been looking at in those pictures. It's how a turtle gets down to where the actual mining is done. Note that `funds`, the very simple *work function* for `plan/snake` is included in the plan definition. There was no need to make a separate library for it. Since the plans are just Lua, we could simply define the function in the plan, just like any other constant.

Finally follow the link in the tree back to `lib/planner` to see how it produces an internal representation for the plan that is used by `lib/worker`.

You can explore the `worker.execute` function and follow the code there backward through the library.

(We've mentioned earlier, given Lua's one pass compiler, that it may be useful to read Lua files from the end to the beginning after looking at either the file's introduction or an available summary.)

As you'll see, the only tricky bit in the code, an admittedly tricky mutual recursion, is the code handling (and potentially recovering from) the things that make life tricky for turtles: getting `blocked`, getting `lost`, and getting `empty` (running out of fuel).

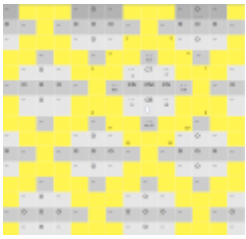
For reference, here are the summaries for these files: for the shaft plan, for the planner , and for the worker.

As we mentioned, The CLL for mining is `lib/mine`. Here's its summary. It supports the remote CLI for mine operations in `lib/net` . The `lib/mine` implementation is straight forward (and frankly kinda ugly) with a lot of status monitoring and error checking as it loads and runs plans. If you must look, it might be best studied working from the end of the file to the beginning.

So far we've only been dealing with the `shaft` command. Actually drilling tunnels (`bore`), navigating in the mine (`post`), and mining the ore (`ores`) is next.

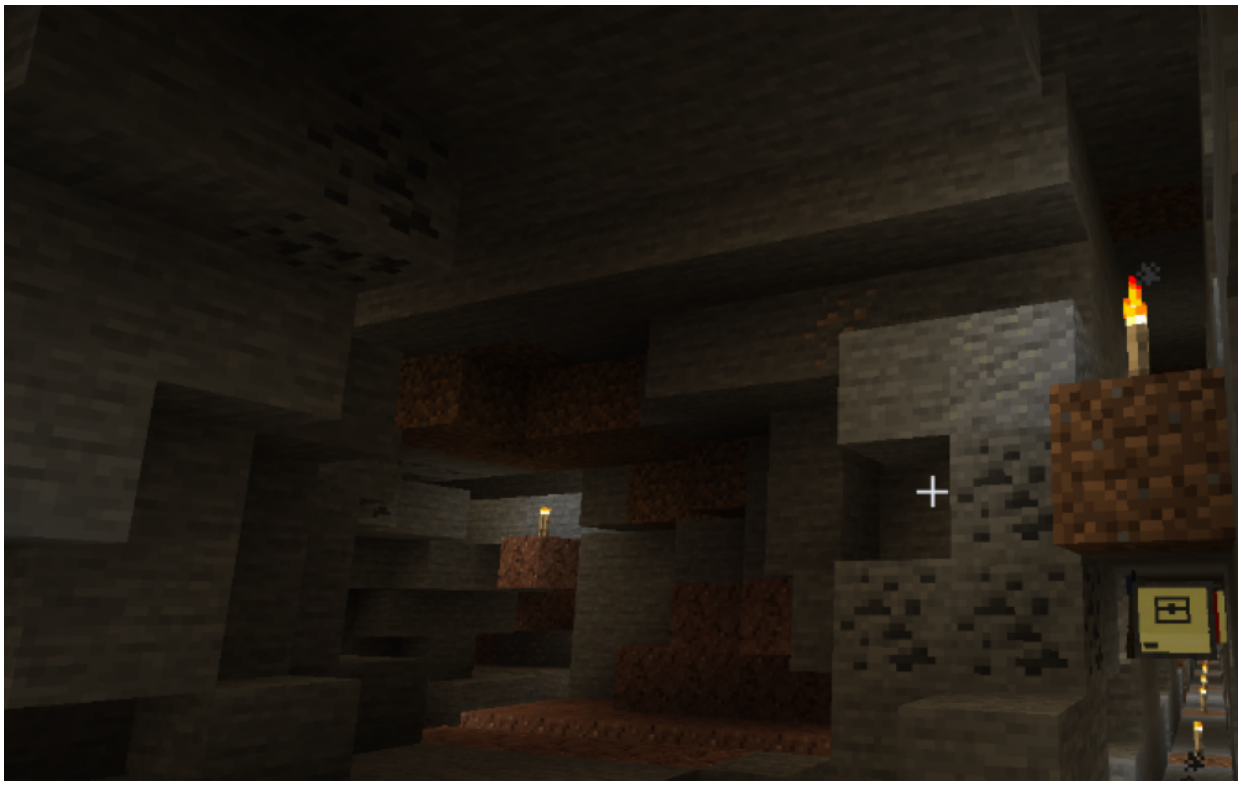
The `plan` for mine shafts get us down to a given level for mining ore (and back again). Tunnels need to be bored at a given level to find and dig ore at that level. To minimize fuel consumption, all the digging at a given position is done while the turtle is at that position. This results in a cross shaped pattern for tunnels. Here's what that looks like at an even numbered level looking south-west:

As you can see in the image above, the tunnel running north provides access to the east-west tunnels where the digging for ore is done. A given for Minecraft (or so we're told) is that ore is never found in smaller than 2 by 2 block veins. We can use that assumption to reduce the amount of digging necessary to find ore. . With this in mind, we can follow the link to look at an excavation pattern that is repeated every four blocks vertically and every six blocks horizontally. (For tunnels as well as shafts, a schematic representation of what is to be done is helpful, perhaps even necessary.) The schematic representation in this case is a cross section looking north, showing how tunnels are to be dug to find ore. The blocks labeled "`O`" are dug in the cross shaped pattern we spoke of earlier. The blocks labeled "`--`" are adjacent to dug blocks and therefore may be inspected as we dig.



One of the repeated patterns is shown with more detail. In that pattern, the dug blocks are labeled "`CT`" (cut top), "`CB`" (cut bottom), "`CN`" (cut north), "`CS`" (cut south), and "`CM`" (cut middle). The inspectable blocks are labeled "`--`" and in the detail they are also numbered. If ore is found in a given numbered block, there may be ore in the 2 x 2 section whose far corner is labelled with that number. We'll dig out that 2 x 2 section. In doing this, we won't dig out any of the other "`--`" blocks since by doing that we would not be able to see if they showed indications of ore in the 2 x 2 veins they were part of. (Yes, it took a number of failed attempts to get this right. Burying this *what* in a bunch of *how* would have been a likely unmaintainable mess.)

Finally, we need torches in these tunnels. They are in put the bottom of the cross shaped dig and replaced after any excavation. The plan needs to work even when tunnels run into caves (as shown).



The *how* that would be needed for any plan for mining in a `grid` pattern is captured in `lib/grid`. The *what* plan for one particular mine geometry, `plans/cross`, references `lib/grid` for the *how* that will be directly handled by the Lua functions there.

Follow the link in the tree to look at the plan for digging, mining, and navigating our tunnels together with the cross section drawing to see how the plan follows from the drawing. The summary of the plan is here.

Note that the tunnels in a `cross` plan form a (rectangular) `grid`. As we mentioned before, the `cross` plan references `lib/grid` so that the plan itself keeps to the *what*. There's an awful lot of *how* in mining. That's kept out of the plan and the *little language* is kept little. There's a library for that: `lib/grid`. As you'll see if you follow the link, it does the most interesting bits.

The next chapter takes the declarative idea to another level. Sometimes rather than defining the `path` of a `plan` explicitly, it's useful to let MUSE generate a plan by just specifying the three dimensional rectangular `bounds` that its `path` should cover. In this case, the support libraries do more so that the client declarations can do less. Because of this, the support libraries wind up being more intertwined with their clients. In particular, control flow passes back and forth between libraries and clients in what might be characterized as a *framework inversion of control* . Before considering the idea of frameworks, we always saw clients calling libraries. The new and surprising thing here is that the library is also calling its clients. Man bites dog. Different dog.

Chapter 7 - Frameworks and Factoring (Again)

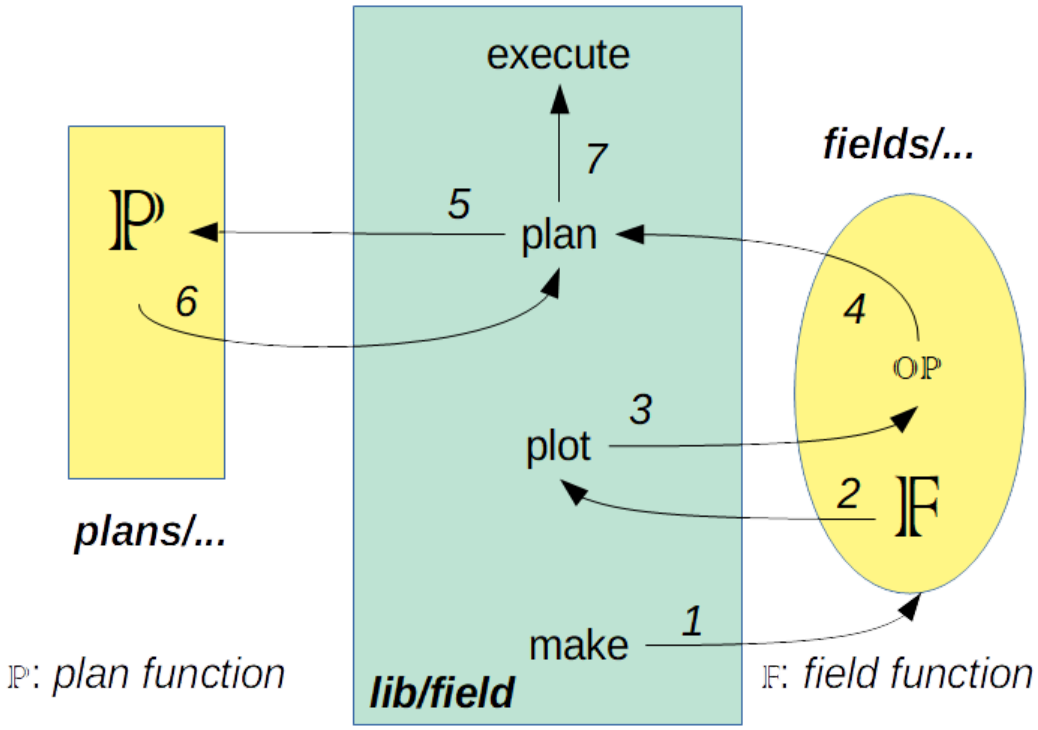
The MUSE framework built around `lib/field` supports work on a *field* as defined by a three dimensional rectangular `bounds`. Each of the files in the `fields` directory is, as you might guess, a *fields file*. Each includes declarative representations of what is to be done in their *field*. One idea they have in common is the idea of a `plot`. Primarily to deal with the limitations of turtle inventory, plots break the field into pieces that can be dealt with separately. Another thing they have in common is participation in a control framework orchestrated by `lib/field`.

As we'll see, there is a complicated flow of control split between `lib/field` and its clients. This complexity opens up many unfortunate opportunities for *bit rot*. So whether to create a framework is a design decision to consider carefully. Sometimes though, it's the right way. And sometimes somebody else has made that decision for you. The blow-by-blow control flow documented below is intended as something you can keep in mind in the event that you are confronted by that somebody else's framework or can't resist creating one of your own.

For MUSE, the hoped for compensating benefit from this complicated flow of control is the ability to use the framework to readily define the *what* for the work to be done on a wide variety of fields because the framework:

- leaves the three dimensional `bounds` geometry of plots in the control of the `fields` file
- while generating an fuel efficient `path` through all the blocks in that geometry;
- supporting whatever suite of operations on the field is established by a set of `fields` files; and
- allowing specification of any `plan` to do the work for the field along the path.

It's not like what we saw in `lib/mine` which knew (in grim detail) all about going down shafts and back up shafts to get to bores which could be used to mine ores. Here `lib/field` doesn't presume much at all about what's to be done in a field. It's just there to (efficiently) conduct the score that it's given. Here's how:



That score is constrained by `lib/field` to fit into a specific set of steps to execute a field command.

- It all starts by loading the client `fields` file by `field.make` (step 1). This produces a function, the *field function*, 𝔽,
- Which is used to call `field.plot` (step 2) with its expected arguments.
- Those expected arguments include an operation function, 𝕆ℙ, associated with the command we were looking to execute. The framework calls that function (as `fieldsOp`), in `field.plot` (step 3).
- That function produces the arguments for the call to `field.plan` (step 4). The arguments include the name of what we'll call a `plans` file for the operation. The `plans` files are in the `plans` directory.
- The music continues when `field.plan` loads that file (step 5) to produce and call the plan function, ℙ.
- The plan function, ℙ given the game coordinate bounds of the field, calls `field.paths` , a utility not shown. This generates an fuel efficient set of paths (in a surprisingly involved computation). It provides these for the framework (step 6) along with the work function for each path.
- The grand finale is when `field.plan` passes control to `_field.execute` (step 7).

There's a `lib/field` utility function, `field.extents`, called by `plans` files which is interesting because it defines a pair of virtual axes, `stride` and `run`. These create an abstraction to define plots in terms that may (or may not) be a rotation of game coordinates. The idea is to do the rotation if that would result in longer (more time efficient) turtle runs along the paths. The extents of the supplied `bounds` of a field are returned as `stride` and `run` coordinates. Field operations (that is,

turtle runs) are done along the `run` of a `plot`. The `directions` for plots are then virtual: virtual north, `vN`, virtual east, `vE`, and so forth.

Making a farm involves a lot of work both out there in reality and here in a game. The land needs to be levelled, the soil prepared, and crops planted. Finally, with the farm established, it can be harvested and, perhaps, new seeds planted for the next harvest. Each of these stages coresponds to each of the the *field operations* defined for a `farm`.

Beating on our orchestration analogy yet some more, the field operations are the movements of the music orchestrated by the set of MUSE `fields` files. For a `farm` they are: `quarry`, `layer`, `cover`, `finish`, `harvest`, and `path`. They're documented in the `lib/farm` library of work functions. All are exposed as remote field operations (say for the `farmer` and the `logger`) as calls to `field.make` by `lib/net`. Nothing in the framework itself imposes these as movements for field operation. They're just what's established by a set of field files.

(There's another field operation, `test`. It's simply a way to do an out of game check that the geometries are correct before plowing up a mess in the game.)

With this overview as context, we'll explore the `lib/field` framework by looking at some `fields` files (for steps 2 and 4) and `plans` files (for step 6) to make this abstraction concrete.

The first `fields` file to explore is `fields/cane`. Declaring the `plots` for each operation on a cane field does most of what's needed. It's just a lot of picky arithmetic to declare the specifics of *what* cane fields need to be. The rest of what's needed in a `fields` file is the definition of the operation functions (𝕆ℙ) executed in support of each field operation.

All these operation functions are pretty simple functions as you can see. They are specified as functions since it didn't seem worthwhile to define a declarative language just for this purpose.

The `plans` files for our framework are like the `plans` files we looked when exploring mines. Except that all its bounds geometry is provided by parameters supplied by `lib/fields` and all its path geometry is provided by its call to `field.paths` and `field.extents` in `lib/fields`. There's not much to it. It's really a plans *prototype*.

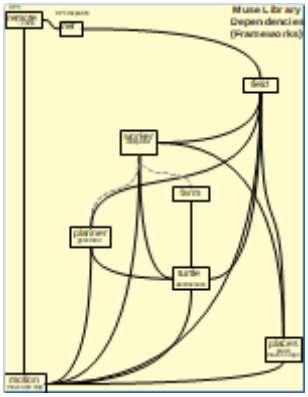
Loading a plans prototype with the bounds parameters from `lib/field` generates that *plan function*, ℙ we spoke of . It calls `field.paths` and `field.extents` to generate a set of efficient paths to work on the volume of a plot. The path generator produces flying ox plow paths through the given three dimensional rectangular bound. Ox plow paths minimize travel to plow a field. Flying oxen (*aka* turtles) do that in three dimensions. That's the only hard part. As we've said, it's in `lib/fields`. Together with the resulting simplicity and generality of `fields` files and `plans` prototypes, it's what we get for getting involved with the framework.

As indicated, there's not much there there in plan prototypes. Look, for example, at the prototype plans/cane file for fields/cane.

The other fields files for farming are similarly conceived: crop, tree, and pen. They make use of `plans` prototypes that look much like the one we've already seen: crop , tree , quarry, layer, till, and path.

The `path` plan prototype is used to traverse the path for a `range` without doing anything else. In the game it provides a way to be a bit more confident that turtles will be doing what you expect when they work the fields. As you may have already seen, it's also exposed as a remote field operation as a call to `field.make` by `lib/net`.

The plan prototype files are all a little different and just the same. This commonality can lead to an unfortunate amount of ill considered boilerplate. The good news is that it's easy to produce new uses of the framework. The bad news is that it's easy to produce mindless stuff that almost works.

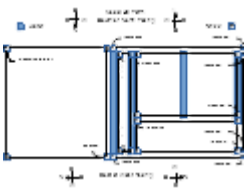


One extra consideration is that many of the `plans` files construct simple `work functions` as we talked about for mines. The `plans` produce the functions consumed and applied by `lib/worker` . These functions may, in turn, use more complicated functions defined by `lib/farm`. Here is a drawing showing how the `lib/farm` and `lib/field` libraries fit into MUSE. They use the MUSE RPC and the RPC dispatch for `lib/field`.

Generating and using this framework involved us in what's been called a separaration of concerns. Of course, libraries by their nature do this too. And we've talked about factoring before. We just have a more complete illustration of this kind of factoring in the `lib/field` framework:

- The `lib/field` module itself is just the definitions of a bunch of mechanisms for doing work while efficiently traversing parts (`plots`) of a three dimensional geometry. It doesn't concern itself with farms and farming. We've used it for that but that's just one possible application of the framework.
- The field parts geometry and a set of appropriate field operations are established by `fields` files. The files don't concern themselves with doing the actual work of farming or how to efficiently traverse the plots of fields. The set of operations does create a common protocol for farm operations exposed by `lib/net`. This is the user interface for farming.
- Given all this, the actual work to be done is determined by `plans` files. Because the plans don't concern themselves with geometry, they can be reused by `fields` files as appropriate.
- Some common operations for the work specified by `plans` files is provided by `lib/farms`. They too have a narrow focus on some specifics of how to do particular tasks.

Each piece of this puzzle has a distinct collection of concerns and exposes just the abstractions that the other pieces need. Making the puzzle gets us reuse of the pieces and focussed maintenance on just the piece needing revision.



There's a final level of abstraction we'll explore. Note that what we'll describe is just a bunch of *what* about making and operating, for example, a farm. There's no *how*. We mentioned `charts` back when we explored `lib/map`. A *chart* is used in what follows to setup the related fields of a farm in a common geometry.

This is laid out, for example in `charts/farm40`. In the example, for each of the related fields, `farm40.lua` creates a *range*. Each range is offset from a common `corner` *reference point* of the farm, the *farm name*, supplied by the user. The `fields` *feature* of this reference point is a dictionary. If a key in this dictionary is a *field name* in the farm, the value associated with such a key is the name of a *range*. The `features.fields` of this range, in turn, is the name of a `field file` (like `crop`, `cane`, `tree`, and `pen`) for the kind of field you might find on a farm (or at least a Minecraft farm). The `features` design idea developed back in `lib/map` lets us extend what's in a `range` and get all the support of a `range` without changing anything in `lib/map`.

For our chart too, just as previously discussed, the `field.make` function kicks off a *field operation*. It expects either the name of a *range* or a *ranged field*. This latter is a string including the name of a *reference point* (like the farm name argument for the `corner` parameter above) and the *field name* of a field associated with that reference point (separated by a colon). Together they specify a range for a field in our charted farm. As described above, the `fields` feature of this range is a dictionary whose keys are names of `fields` files. The work for a *field operation* is found in that file.

The `charts` idea only concerns itself with the layout of a farm. It hides a lot of complexity by building on the framework without exposing it. All that framework orchestration is on the other side of the curtain. The mechanisms we've been describing just work on what's defined by a chart. It's the final payoff for dealing with this framework.

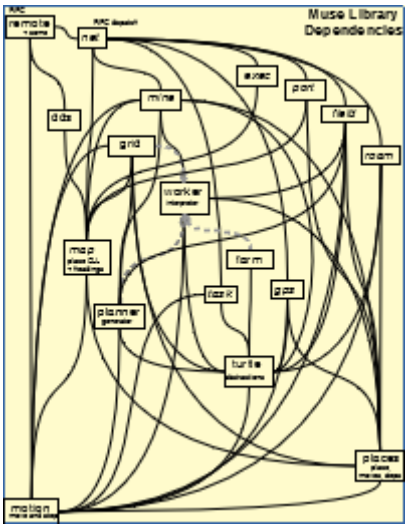
As a design idea, the `charts` facility is (scarily) open ended. It's the conventions around its use that make it work. The approach is powerfully unconstrained. It's another power tool. Watch your fingers. There's nothing enforcing (or even supporting) the conventions. Here, again, be rope. Not that there's anything intrinsically wrong with rope.

One final note as we conclude. There are a few utility functions we can just point at in `lib/field`. They support simple operations with simple implementations on volumes defined by pairs of points or a range as named places. Simple operations to `cut`, `fill`, `till`, and `fence` don't need the complexity of a framework (though they do make use of `plans`). Kind of a relief really, after what we've been digging around in.

Chapter 8 - So This Is Goodbye

If you've gone along on this ride, you've been with me on an exploration into some of the ideas and implementation approaches for more maintainable code. It may seem like over engineered work. Of an obsessive compulsive engineer. With way too much time on their hands. (After all, it's just a game.) Perhaps. But, you know, it's never been about the game. At any rate, I hope it will have proved useful to you, maybe even (someday) in the deadline fevered heat of a moment. Maybe you'll just do nothing, stand there for a bit, and think about those AIs from another planet. The ones we met when we first started down this road. You will meet them again.

Together we've looked at mindfully managing state because there, there be dragons. We've seen how structuring code in libraries can isolate dependencies and create layers of abstraction. All this so design, testing, and maintenance can focus on what's changed when dealing with changes. We looked especially at factoring the code to anticipate (the inevitable) changes to the user interface.



We (or at least I) have had fun with functions. We've dealt with state distributed across a network and persisting over time. Distributing state across a network led us into the issues of concurrency, service discovery, and remote procedure calls (that need to deal with error conditions).

As part of a developer's bag of tricks (some more magical than others), hey presto, there be declarative little languages and frameworks. If you need them, there they are.

The code we've looked at is peppered with documentation. In MUSE, given its goals, this kind of seasoning is particularly heavily applied. However, most code profits from some such. Using CodeMark for concise and expressive code annotation, MUSE gets support for code completion and tool tip supported editing. It does this for a couple of development frameworks. It takes advantage of what type checking there's available for Lua by integrating with LLS, the Lua Language Server. Those libraries we went on (and on) about get summary level markdown and HTML documentation of what they do and the interfaces they support for doing it.

Following all those hyperlinks, there's been the opportunity for many field trips outside MUSE. I hope you didn't get too lost.

The code for MUSE is part of what's supplied (under an MIT license) for you to use, experiment with, and evolve. There are `TODO:` comments throughout the code with suggestions for that. All with the intent of learning to develop maintainable code. I'll be interested in what you do with it and to it.

Code well.

Appendix: MUSEcraft - Running MUSE

When you're ready to try running MUSE in the Minecraft/Computercraft environment, come here for some help in doing so. We'll look at three stages of help: setting up Minecraft for MUSE, getting started with a ComputerCraft world for MUSE, and making changes to MUSE.

Minecraft to Start With for All the Worlds You Create

- First, you'll need a licensed copy of Minecraft. We've run MUSE with Minecraft 1.20.1. Select that version on the Minecraft launcher and run it once so you can install the rest of the Muse environment.
- The MUSE environment depends on Minecraft plugins. Use Forge to install them. Double click the `.jar` file to install the Forge client in the .minecraft directory then put the following plugins in the `.minecraft/mods` folder:
- CC: Tweaked 1.108.4 for Computercraft itself.
- Global Packs so that you can use Muse in all the worlds you create.

- Just Enough Items 12.1.1.13 (which is helpful rather than necessary).
- Then create a folder named `MUSE` in `.minecraft/resourcepacks`. Download and unzip a MUSE release and copy the contents of the MUSE release below the release folder itself into the `resourcepacks` `MUSE` folder.
- Create a MUSE directory and set an environment variable, `MUSE_PROJECTS` to its path. (For Windows, use the `Advanced System Settings` in the `System` Control Panel)
- For Lua's `require` function, set the `LUA_PATH` environment variable to include:

```
%MUSE_PROJECTS%/CodeMark/?.lua;%MUSE_PROJECTS%/MUSE/data/computercraft/lua/rom/modules/lib/?.lua;;
```

- In the Minecraft launcher, choose the Forge 1.20.1 installation and ask to `play` it.
- When the loader finishes it's work, choose the `Options` in the Minecraft splash screen. Open `Resource Packs`. Look for `MUSE` (and `computercraft`) among those available. Hover over and click the arrow to include in those selected. Click `Done`. Back among the `Options`, click `Done`. Back in the Minecraft splash screen, Click `Single Player`.
- Create a Minecraft world and launch it. `Open to LAN`` and be sure cheats are enabled. You may prefer a "peaceful" level of difficulty. The seed to create a Desert Village world is 1342300981. The seed to create a Forest Village world is 2583372187968070576.
- It may be helpful to use a viewer to find a place for your base. Perhaps somewhere reasonably flat to farm and near a village to trade. Check it out safely using the `creative` gamemode.

Next: ComputerCraft (Tweaked) and MUSE for Each World You Create

Setting up the GPS is one of the first things to do in the Minecraft world you created. The coordinates for the GPS you set up need not be aligned with Minecraft coordinates, but it helps in using some game facilities if they are. MUSE provides a way to do this using a command computer. You'll need to enable its use in survival mode. First up, change the configuration settings and increase the storage space on ComputerCraft computers. On Windows, the file to change default settings is at:

```
.minecraft/defaultconfigs/computercraft-server.toml
```

(We'll use forward slashes in file paths for consistency with non-Windows systems.)

The settings to include in the configuration are:

```
computer_space_limit = 100000000
command_require_creative = false
```

There's a `computercraft-server.toml` file in the `MUSE/data` directory that you can copy to the `.minecraft/defaultconfigs` directory. These 'default' settings seem actually to override the settings for all worlds. (If this doesn't work, each world's settings are in the world's `saves` as `.minecraft/saves/<world>/serverconfig/computercraft-server.toml`.)

Doing much of anything with MUSE turtles will need fuel. For game experience, you can play to acquire fuel and maybe a crafting table (or use the coal supplied by `muse:base`).

With a fuel supply, you can go about setting up (the required) GPS navigation for MUSE. You'll need a pocket computer (the `player`), a command computer (the `porter`), a turtle (the `rover`), and four computers to provide GPS facilities. The GPS computers will need disk drives and a floppy disk. The turtle will need tools. All this lot will need modems. So many, many parts. Running `/function muse:base` provides all these parts in player inventory, sets the world spawn, and turns off Minecraft `DaylightCycle` and `WeatherCycle` (as a convenience).

The next step is putting together all those parts to craft what you'll need for the GPS:

- Craft the pocket computer with an ender modem on top.
- (This will be an Advanced Ender Pocket Computer.)
- Craft a turtle with an ender modem on either the left or right side.
- Craft a pick axe on the side opposite the modem on that turtle.
- (This will be an Advanced Ender Mining Turtle)

You will need to go to `/gamemode creative` to place the command computer. Place it somewhere convenient with open sky above it. Then go back to `/gamemode survival` and `sneak` to place an ender modem on its back.

Select and (right) click the `player` pocket computer to `use` it. Run the `peripherals` program to make sure it has a modem. Just as a check, click to `use` the (`porter`) command computer and run `peripherals` on it as well.

The rest of the given `base` inventory will be used to `launch` the GPS computers. We'll get to that in a bit. First, `reboot` CraftOS. This registers what you've placed in your world with the MUSE Distributed Discovery Service (DDS) for MQ hosts. There should now be two DDS hosts: `player` and `porter`.

The reboot involves a bit of indirection. The `muse.lua` file in the `MUSE/data/computercraft/lua/rom/autorun` directory is deliberately dead simple:

```
--:~ Autorun -> **Enable access to MUSE programs and start MUSE** <-
muse/docs/autorun/muse.md
shell.setPath(shell.path()..":".."rom/programs/")
shell.run("rom/modules/.start.lua")
```

The reboot runs the `.start` daemon to establish and report the MUSE game environment and then puts the `porter` and any turtles (that you'll create) in a `MC` (MUSE Call) rednet protocol wait loop. This is the RPC wait loop that you saw in `lib/remote` .

When the reboot is done, set up to align the GPS coordinates with Minecraft coordinates by placing the Advanced Ender Mining Turtle you crafted on top of the `porter` command computer.

When you mouse the turtle, a number should appear above the turtle as its "nameplate". This is its temporary `label`. Let's say it's the number `2`. To assign the role of `rover` to the turtle, run `join rover 2` from the `player` pocket computer (or whatever number appears as its nameplate). This enrolls the turtle in DDS. It will be known as the `rover` after the next CraftOS `reboot`, which you should do now..

After the reboot, the turtle's nameplate should read as `rover`.Then run `locate` from the `player` pocket computer to name the place above the `porter` where `rover` sits and determine `rover` orientation . Let's say you named the place `stage`. Once `locate` has run, move the remaining player inventory items into the `rover` inventory. Make sure there's clear sky above then issue the `launch` command (as, say, `rover launch stage`) with the named place you setup by running `locate`. The GPS computers will get set in space and have their own startup files that don't enroll them as `MQ` hosts. After reboot, GPS should now be active. Setup is complete. But there's more.

Muse provides support for the idea of a `site`. There may be several of these in a Minecraft world. Each could have a set of (so called, `landed`) workers for that site: a `farmer`, a `logger`, and a `miner`. Running `/function muse:site` as a cheat puts the turtles for a `site` in player inventory with the needed modems and work tools. Sites are named using the, ahem, `site` command with a unique name for the site. When places are named, their names are qualified with the currently established site. When `join` is run to establish landed turtles turtles in the DDS, they will be given unique, sited roles (and labels) with the currently established site. This is important since `rednet` hosts need unique names. If you've named places, you can run `sync` from the `player` pocket computer to include those places in the new turtles' `map`.

When setting up a new site, you may want to keep site workers working while the player is elsewhere. Use the `activate` command with a MUSE `range` encompassing the site to establish the site bounds to keep turtles active while the player is away. (It seems, however, that this won't keep crops growing.)

That's it for getting MUSE running in your Minecraft/ComputerCraft world. To see what you can do with MUSE you may want to look at the programs summary and the remote call summary .

If you want to experiment with changes to the MUSE implementation and its documentation, that's next

Visual Studio Code for Source Control Management, Syntax and Type Checking

VSC's integration with GitHub makes source control pretty easy. With its help you'll have a remote repository for your code if the not so unimaginable thing happens to access to your local copy. So, first things first, get started with `git` and Github:

- Educate yourself with the very basics of `git`. There's a free ebook. No need to go deep just now. Learning just what you need as you need it will be effective. Then get yourself a GitHub account and install VSC and the extension for git. Close VSC.

- Use VSC to clone the MUSE repository into the project directory you created.
- If there's a MUSE directory in `.minecraft/resourcepacks`, remove it. You'll use the (potentially edited) cloned copy instead. *(We'll show forward rather than back slashes for file paths as a bow to industry consistency.)*
- The cloned copy of MUSE is in the project directory. Create a symbolic link to it in `.minecraft/resourcepacks`. (For Windows, there's a GUI, Link Shell Extension, that may be helful.)
- Use VSC to clone the CodeMark repository in your project directory. (So you can keep code completion and type checking current.)
- To take advantage of Lua code completion and type checking in VSC you'll need the `sumneko.lua` (Lua Language Server) extension.
- If you change `MiningMuse.md`, you'll also need the `yzhang.markdown-all-in-one` (Markdown All In One) extension to recreate its HTML file. After it's loaded, go to the VSC settings for Markdown CSS and add `markdown.css` to the markdown style paths.
- You may want a couple of additional extensions: `MinecraftCommands.syntax-mcfunction` and `be5invis.toml` (TOML Language Support).

There are VSC Lua debug extensions. It would be good to know if they work for what you are doing. Or you could just use VSC for its syntax checking, type checking, and source control management support while using Zerobrane Studio for your run and debug environment. That's next.

Zerobrane Studio (ZBS) for a Run and Debug Envronment

There's not much left to do in setting up your development environment for MUSE but it's arguably the most important part. Making changes to MUSE will mostly involve running code and testing it in the Zerobrane IDE. VSC analysis and source management is critical but not the main event. So, on to the main event:

- Follow the installation directions for Zerobrane Studio
- Copy packages : `analyzeall.lua` and possibly `cloneview.lua` into the ZBS `package` directory
- Create a symbolic link from `CodeMark/apiMark.lua` in your project directory to the ZBS `package` directory. You'll be able to update (but not remove) ZBS code completions for the file you're editing directly from ZBS in the `Project` drop down menu.
- Create a symbolic link from `MUSE/data/muse/signs/muse.lua` to ZeroBraneStudio's `api/lua/` directory.
- Edit (or create) the `cfg/user.lua` file in your ZBS installation to include `api = {'muse'}`. You might wat to look at the documentation for information about ZBS configuration.
- Set the project directory to the MUSE folder in your project directory.
- Open `MUSE/data/muse/assets/museMark.lua` and execute it (without debugging). This should produce all fresh, all new, all the code completion, all the type checking files, and all the code documentation for MUSE.

There are tests for MUSE but no test framework. MUSE tests rely on human review of their output. A framework automating tests would be a useful addition (as would more extensive testing).

And that's the lot. There's an issues reporting page for MUSE if (really when) you find something that needs reporting. All the best.