WanderBits Work Log

Friday Afternoon, Sept. 27, 2013

The project is almost done. I will spend a few minutes right now hooking up the last parts of the `Look` action. Then I will update the unit tests. Then see about a few more actions. Then put together a writeup.

I finally got the Look action working. I have added silly placeholder actions for `Put` and `Take` actions. I'm going to work now on a description of my design decisions. This log file should help out a lot.

Friday Morning, Sept. 27, 2013

This is the last day for the project. So what is the next big thing on my list? I made a start last night with the container feature. I realized then that my ideas about **Intimate**, **Local**, and **Global** scope might be too complicated. Or perhaps they are more of a guidance, rather than concepts that must be explicitly coded into the game. Going forward, I am going to focus on the idea of each `Thing` having a container property that allows it to hold other `Things`. The methods `Add` and `Remove` will make this work. I still wonder about making a transaction happen. Maybe I can add the object in question to the new `Thing`'s container, then so long as there were no errors, I remove it from the object's old container. In order to make this task easier, I will also store with each `Thing` a link to it's parent object, i.e. it's container. This link must be updated as part of making a transaction.

Thursday Evening, Sept. 26, 2013

I just added new unit tests for the major features I added today. I'm getting pretty tired! I connected the response output from each action to the user's console display. The next big feature to add is the concept of a `Thing` being contained inside another `Thing`. This will enable the `User` class to have a sense of location in a room. The first `Action` to hook up after the container feature will be the `Look` action.

Thursday Afternoon, Sept. 26, 2013

This afternoon and this evening is a big push. Ok. This evening it feels like I am rushing through my work. I've done lots of detail work on Actions, Things, and Executive and I really want to take time *right now* to work on some unit tests. I feel pressure to keep working on more features! Time for a feature break. I'll add a few unit tests and then regroup on the next feature.

Thursday Morning, Sept. 26, 2013

Here we are getting close to the end. The only class I have yet to make progress on is the `Thing` class. This class is also potentially the most complicated of all. An abstract `Thing` has relationships with other `Things`. The nature of the relationship defines restrictions on whether or not a given `Action` has access to a `Thing`. Relationships may or may not be reciprocal. If `Thing` A is within the local scope of `Thing` B, then `Thing` B is also within the local scope of `Thing` A. However, if A is inside of B, the reverse is *not* possible. Occupying one scope may also imply occupying other higher-level scopes.

1. **Intimate**: This scope is used to convey being attached to another item, inside another item, or on top of another item. This implies a physical connection. A `Thing`'s physical location is determined by it's intimate relationship with another `Thing`. An apple `Thing` may be within the intimate scope of a box `Thing`, the box `Thing` may in turn be within the instimate scope of a room `Thing`. An object may only occupy one other object's intimate scope. Multiple objects may occupy a given object"s intimate scope. If A is within B's intimate scope, then B is defined to be within A's *local* scope. Intimate scope is not reciprocal.
2. **Local**: This scope is one layer up from the intimate scope. This scope does not imply physical contact, but it does imply being observable. One may physically manipulate (e.g. take, eat) other `Things` within the local scope. Local scope is reciprocal.

3. **Global**: This scope is automatically implied between all existing in-game `Things`. If a `Thing` is destroyed, it will no longer exist within any other `Thing`'s global scope. Global scope is reciprocal.

Wednesday Afternoon, Sept. 25, 2013

Before going further with `Executive` I am going to implement an `Action` class. What is an `Action`? It controls how change happens in the game. The user enters a valid command, and this is translated into an `Action`. The arguments that follow the action verb on the command line are given to the matching `Action` instance. The `Action` in turn then does whatever is needed to realize the user's command. The arguments correspond to: objects that are within reach of the user, allowed navigation directions, more?

There could potentially be a large number of game `Actions`. I think now that it will work best if each kind of `Action` inherits from a base `Action` class. I am also going to take this opportunity to use the Python Abstract Base Class feature. I've seen it before but never used it. This seems like a good time!

Wednesday Morning, Sept. 25, 2013

Working a bit at home before work.

I thought more about the `Console` class. I don't think I need it as all it does is handle reading/writing from/to stdin/stdout. I think it will be much simpler to move the read and write functions over to the `Executive` module. More tidy implementation.

While merging `Console` stuff into `Executive`, I noticed that I could add one last detail to `Parser`. Right now it simply returns a list of tokens. From the beginning I have been thinking that my command language is made up of a leading action word followed by one or more arguments. I will mod the `Parser` such that it outputs two things: the action name and a list of action argument names. These actions names will not yet be validated against the game content. That part will happen inside the `Executive`.

Tuesday Morning, Sept. 24, 2013

Today I have a big meeting Santa Barbara for a big work-related meeting. I got here in town a bit early so I get work on this little project a little bit at the Starbucks.

So far I have the `Parser` and `Console` implemented with basic functionality. The `Executive` will be the one talking to these two parts. The `Executive` needs to send messages to the `Console` and receive user input from the `Console`. Soon I will have to get working on `Things` and `Actions`. But before that I can hook up the `Console` and `Parser` to the `Executive` and run a simple event loop.

Ok, I got a few good bits of work done with the `Execute`. The event loop works and it can echo back to me the text that I type in. Command prompt is simple and neat. That's all for this morning.

Monday Morning, Sept. 23, 2013

I have just a little but of time here before I have to take my son to school. He's playing Scribblenaughts on the Wii right now. Its hard to get work done as he's asking me questions every couple of minutes.

I need to clarify how the `Executive` manages connections between components. The most direct approach would be for `Executive` to be initialized with all necesary startup information. This could be names of config files. `Executive` would then be in charge of instantiating appropriate `Parser`, `Console`, `Things`, and `Actions`. The details of this part are not completely clear. Anyhow, at some point the `Executive` must be fully configured and ready to start running the game for the user. The `Executive` should have a `start()` method that kicks off it's event loop. This method will block until the user's game session is over.

I just wrote the `Console` class. There is not much to it. I really might not even need a seperate class for the `Console`, but it at least segregates the particulars of stdin and stdout from the game's inner workings.

Sunday Evening, Sept. 22, 2013

Basic `Parser` is now writen.

Hmmm. I just thought of something. The `Parser` currently checks through the input tokens for valid argument `Things` after the `Action`. This might be a mistake. Each `Action` really has its own set of valid `Things` it can work with. It will be easier to have the `Executive` handle making these checks. Let the `Parser` take care of clobbering punctuation and known ignore words. That will make the `Parser` definitely more focused. I also found out that the package name 'parser' is already used by Python itself. Hhhmm. I think I have the parser name thing figured out. It should be ok as long as my parser stays within this game package and I don't try to import it interactively from outside the package.

I finished implementing a good start to the unit test for the parser module. I'll more tests to it as it develops.

Sunday Afternoon, Sept. 22, 2013

No work done this morning. Had to go a kid's birthday party. Now it's just after 2pm and I plan to get a lot done with the `Parser` class and make at least a start on the `Executive`.

The `Executive` can start off as whatever script I build for developing the `Parser`. Here is how I'm thinking the data will flow: - start up - executive ataches input stream to `Parser` instance - `Parser` waits for new line of text from user - Parse each line according to the game rules, this will be the major work area for today - `Parser` will be a generator yielding parsed results back to the `Executive` - `Parser` keeps parsing until told to quit by `Executive`, or ctrl-c.

Ok, let's make that happen.

Some Ideas to Think About

1. I will need something that handles the text IO with the user's console. I don't think the parser should be doing this job. I'd rather `Parser` did its one parsing job and nothing else. This new thing could be called a `Console`. It would know about reading from `sys.stdin` and writing to `sys.stdout`. More details about `Console` later.

2. The `Parser` needs to know some information about which `Actions` and `Things` are valid for the game. But I do not want the `Parser` class to be dependent on my implementation of `Thing` and `Action`. I think the software will be more robust if the `Parser` is told about the game via a list of strings for the names of valid `Things` and `Actions`.

3. Lets say that the `Executive` will handle processing data related to `Actions` and `Things`. My data flow might look like so: `Console` –> `Parser` –> `Executive` –> `Console`. Any given user input will start at the `Console` and ultimately end at the `Console` when the response text is displayed. Each downstream component is waiting until the upstream components finishes its work and passes it along. The dependent components could be connected a number of ways. In my mind it seems natural to use generators to enable the downstream components to simply iterate over user commands. I know it could be equally implemented using coroutines, where the information is actively pushed to the next worker in the pipeline. Generators seem an easier approach here.

Saturday Evening, Sept. 21, 2013

It's time to think about how to incorporate Unit Testing. So far I have a `Parser`, an `Action`, and a `Thing`. The `Parser` might be the simplest of the three. Or at least right now I think I can visualize in my head all that it needs to do. Hmmm, perhaps I should write down what those things are. And then won't that set me up to write Unit Tests? Curious.

Parser: I could have some predefined sentences and confirm that the `Parser` parses them correctly. This one seems straightforward enough.

Thing: I described earlier some of the features that a basic `Thing` must have. I could test the ability of putting something inside a `Thing` and then taking it back out. Test whether a `Thing` inside of another `Thing` is visible or not. Count how many items are inside. Make more tests to cover new features of derived subclasses.

Action: `Actions` make stuff happen. A simple test would be to set up a game scenario and then make an `Action` do its thing. Verify afterwards that the task got done. That's all I can think of right now. More later.

Saturday Afternoon, Sept. 21, 2013

It feels like a slow start. I have lots of ideas in my head and they are swirling and swirling. I don't yet see clearly what the major components are going to be nor do I see how they will connect together. I going to write next a few paragraphs describing the game.

The Story

WanderBits is a game of adventure in the classic sense. Remember Zork? I sure do! I remember learning about it in high school and being completely mesmerized by the entire concept. It was like reading a book with which you could interact directly. I could never get enough of those games!

The game of WanderBits is the story of a young man on a quest to explore his world. We don't know the name of this person, or where he comes from. The world he lives in right now is a bit of a puzzle. It's made of up curiously-connected rooms, but some of the doors are locked and with no key in sight. There must be a key somewhere around? Maybe in that pot over there? Maybe under the carpet? Hmmm??? We must help this poor fellow find his way out of this game so he can continue on with his life!

Ok, I think that is enough for story time.

Game Parts

A text parsing system can be made as complicated as you can imaging. The first thing I thought about was the Python-based Natural-Language Tool Kit (NLTK). I've always been curious about it and I have wondered what kind of cool things I might do with it. Shit! Forget about that. Back on topic. The scope of this task is such that I need to keep stuff simple. Game commands will be defined by an action word plus one or more arguments. It will be much simpler to implement if the number and kind of arguments for each command are static. Shit. I just thought of something: does the command language also need to specified via config files? I sure hope not... Well, I just read the email again:

> The game engine should be generic; as much as possible all the game specifics should live in the config file (description strings, supported verbs, room layout).

It says *supported verbs*. One way to interpret this is so: the game engine comes with predefined actions, the config file may specify which actions are available in a given game instance. The config file could also specify any number of aliases for each aaction. Another way to support this requirement is to write actual Python code in the config file that implements a particular action. That would not meet the other requirement for the config file to be useable by a non-expert user. I will assume for now that this interpretation is fine.

Lets talk about a few important parts that make up the game engine.

1. **Parser**: A command parser for this type of game needs to support a relatively small number of basic functions. It needs to know about actions for navigating and interacting with the game. It needs to know about things to which the actions are directed. A given command may involve multiple objects, e.g. put key in box. Basic functionality of the `Parser` should be to receive as input text from the user after they presse the Enter key. Idea: parse the text into a list of tokens using something basic like Python's shlex module. Search from the start of the list for a word that matches the name of a command (including aliases). Once found then begin second part of selecting the arguments. White space is no problem. But what about connector words like "the" or "a"? e.g. "Take the apple" should be equivalent to "Take apple" and "Hey, take that apple" or "ehh, grab them apples". I could maintain a list of ignore words and simply skip over them while parsing the user text for commands or arguments. I like that idea a lot, but is it too simple?? I can also make aliases for the plural version of objects by adding an "s" to the end of each object's name.
2. **Actions**: I really like the idea of having a fixed set of arguments for each action command. I can define a base `Action` class that implements basic action capabilities. Then create subclasses for each in-game action. In order to perform its work, the `Action` instance will need access to local in-game items. This could be done by searching through `Thing` instances, starting with the user `Thing`, and then other local `Things` for suitable matches to the argument list. An `Action` makes stuff happen by manipulating in-game `Things` represented by the input arguments.

An `Action` must be able to find the proper `Thing` instance(s) that are refered to by a given input argument. These `Things` must be in the local area and also be visible, otherwise the action cannot take place. For example, 'Take apple' may be interpreted as a task to be performed by an instance of `Take` (a subclass of `Action`). `Take` could have a method such as `do_it()`, or `Take` itself may implement `__callable__` and thus be called directly with the input argument. In this case, then input was the word 'apple'. The `Take` action must be able to search the local area for an instance of `Thing` named 'apple'. Once found, `Take` would first ask the apple `Thing` to remove itself from its current container. Then the `Take` action would add the apple `Thing` to the user `Thing`'s container.

3. **Things**: In-game content is to be represented by objects subclassed from a `Thing` base class. Items like an apple or a key could be represented by a `Thing`. But it gets better. A room could be a `Thing`, and even the user could be a `Thing`. A `Thing`-based object would know about concepts such as what other `Thing` is it inside of? A room, a box, or the user's pocket? The act of moving the user from one room to another would involve moving the user `Thing` over to the inside of the next room `Thing`. The basic `Thing` class should handle stuff being inside of other stuff. This means a `Thing` must know about the sizes of `Things` and the size of its own insides. A `Thing` can be destroyed. The set of `Things` inside a `Thing` may or may not be visible to the nearby user. This allows for a box to be opened or closed. Only `Things` that are visible may serve as valid `Action` arguments. A `Thing` is able to describe its appearance when the user looks at it.

4. **Executive**: All the stuff above needs to be attached to something and somewhere there needs to be an event loop running around. Right now I'm thinking there could be an `Executive` module that is the central place where everything comes together. It might be a simple module with a bunch of functions, or maybe it should be its own Class, perhaps yet another kind of `Thing`? This executive could also be responsible for reading text from the user, and then also printing responses back to the user.

Saturday Morning, Sept. 21, 2013

I also need to include unit tests. That will slow me down. Might never get to the monster :(

Write out an example game session. Make sure to illustrate key types of actions and commands. Think about a simple grammar for parsing user commands. I could build a concise dictionary of verbs, nouns, maybe adjectives and adverbs, plus connector words.

Classes to define in-game objects, and how objects interact with each other and the user. Idea: start with class Thing from which all others inherit: items, rooms, and the user! The config file as a serialization of all in-game Things.

Friday Evening, Sept. 20, 2013

It's time to write a text-based adventure game! Wooo!

Should there be a sense of time? Would only make a difference if anything in the game had a temporal dependence. Like another entity in the game! A monster moving about could be neat, but it would also take time to implement. Hmmm, maybe stuff like that should only be considered after the basic stuff is in place. But still thinking along those lines: user has to search for a weapon while the monster is moving at random (or with a pattern?). Very simple combat implemented as actions connecting weapons in possession with monster. Some items might serve more damage than others. The user and monster have finite amounts of hit points. Turn-based attacks. But if user waits too long, monster will go ahead and eat your head off. Describe monster as horribly grotesque like Cthulhu. I really like this Monster idea, but I must absolutely get the core features implemented first. Monster is bonus fun stuff I get to do after the work part is done well.

The main objective here is to demonstrate programming skills. Tying in to fancy external libraries may not help me get further along here. Some external libraries would certainly add nice features, but all that would do for me is demonstrate how well I can learn to use a foreign piece of software. It is more important to write my own features from scratch, even though they will certainly not be as capable as something I might find somewhere online. Forget about NLTK, forget about an AI bot, forget about a monster that will hunt down the user.

I can implement on my own all of the above required features. It might be a boring game, but the instructions say nothing about making a fun game! Ha! Is that a loophole? Maybe not. Nevermind. I will go crazy if I try to think or predict all the possible things they might be expecting. I have to take the written instructions at face value.