Daily Scrum (Ignore this if you are reading it)

1. What I did?
   1. I did some more Python practice
   2. Read some of Headfirst Design Patterns
   3. Installed NetworkX and created a few simple graphs
   4. Watched a few videos on Maze creation
2. What am I going to do?
   1. By the end of the day, I plan on getting through to at least the end of Chapter 4 on Design Patterns
   2. I will try to help my team create commands that would allow the user to interact with our game
   3. Finalize story line
   4. Figure out a parser? NetworkX implementation of Abstract Factory or both?
   5. Saving the file, what if the user has to exit and want to go back to their save file?
3. Do I have any blockers?
   1. I need to see the API

6/12 Notes (Powerpoint Notes start at Page 2)

Python is built around dictionaries. Everything in Python is just a whole bunch of dictionaries.

Verb object pairs and do stuff with them.

Python allows method references. You can pass functions as arguments. First-class objects. (Ex: def foo is a value). You can return it at a result. It can be a value in the dictionary.

Leverage Python’s strengths. Find out what the language is good at and use them. 2 strengths of Python: dictionaries and method preferences

For our game:

The verb.object pair is the key.

The method is the value in the key-value pair.

I think we will just do multiple keys with the same values for same actions or use lists in dictionaries as Rachel has done

Non-functional Requirement Examples

* Performance
* Real-time performance
* Maintainability
  + Readability
  + Modularity
  + Debuggability
  + Traceability
* Modifiability
  + Lots will change

Analysis

* Static – analyzing the (non-running) code (BEFORE IT RUNS)
  + Linters, cyclomatic complexity (number of possible paths through your program), method length, etc.
  + Pylint, Sonar Cloud (Static analysis tool, does not run your program (checks program before they run)), radon, Checkstyle
* Dynamic – analyzing a running program (We won’t look too much about dynamic analysis)
  + Spotbugs (used for Java), Valgrind, Purify
  + Looks at your program as it is running and looks for mistakes
  + However, for this class, we are going to do a lot of static analysis, not dynamic analysis.

Testing

* Many types (These are the major types of testing)
  + Acceptance testing (The functional requirements of the customer are met)
  + Unit: method (We will do a lot of unit testing TDD)
  + Integration: class
  + System

TDD – Write a test before you write any functionality. If you can’t test your code, you can’t possibly create your code. If you can’t test it, don’t write the code because you don’t understand what you are supposed to do. The test has to fail at the beginning, and write just enough code for that test to pass. You want to commit for the code you have written that has passed the tests. YOU WANT THIS ON YOUR RESUME. Instead of asking yourself, how do I get it to work, ask yourself how do you make it break.

Alpha testing – testing internally

Beta testing – Give it out to a select community to test

Unit Test

* Arguably the most important
* Many modern SE methods use “test first”
  + Write a test
  + Make sure it fails
  + **Write just enough code to make it work**
* If you don’t know how to test it, how can you write it?
* Correct frame of mind
* Test-Driven Development

Write as few lines of code as possible to prevent bugs.

Implementation

* SE tends to focus on requirements, design, and processes
* A bad implementation will ruin everything else

Guideline: Style

* Follow language styles
  + E.g: The Elements of Java Style by Scott Amber and Trevor Misfelt
* Pep8 for Python

Manage (Manages the people and finds out how they work best)

Leaders find out what makes us all the same (gets us excited for the project, puts us in the right **direction**)

Guideline: Naming

* Use good, descriptive names
* Probably not Hungarian notation
  + Type checking now done by IDE and compiler

Guidelines: Test

* Every method has unit tests
* Rule of Thumb: You have written as many lines of unit tests and actual code
* Tests must be easy to run and run a lot
  + E.g.: Junit
* TDD

Guideline: User Libraries

* Developed and tested by many others
* Don’t re-implement
  + You need to learn what is possible
  + Yes, you may know how and think you’re better, but you’re not
  + Write something that no one else has written

Keys in a Dictionary form a Set: TRUE

Values in a Dictionary form a Set: FALSE

Guideline: Reviews

* Code reviews are very effective for improving quality
* Knowing that someone else will be looking at you
* Others know things you don’t

Guidelines

* We’ll be covering other guidelines such as DRY, YAGNI, SOLID, Liskov substitution principle, etc.

Meetup > djug meets at Pivotal labs every month

DRY – Don’t repeat yourself (don’t repeat your code)

Agile

* Many different processes that emphasize short release cycles and that allow changes in design
* Focus on human and team aspects

Requirements drive the architecture of the program. An architecture is above a design pattern. An architecture includes many designs. Design pattern is one layer below architecture.

Agile

* Short releases and iterations
* Incremental design
* User involvement
* Lightweight documentation
* Informal communication (out standups, etc)
* Change (When the user changes their mind, we will do it and the ramifications of what they want, the user is always right)