CSCI 362: Final Report

Testing NetHack

TeamName:

Daniel Hwang, David Rust, Kyle Stewart

TeamName:
Daniel Hwang
David Rust
Kyle Stewart

Table of Contents:

2 - **Introduction**

3 - **Chapter 1: Process to Compile Nethack for Ubuntu**

4 - **Chapter 2: "NetHack Test Plan" by TeamName**

5 - **Chapter 3: Testing Framework, Test Cases, and How-To**

7 - **Chapter 4: More Test Cases, Experiences**

8 - **Chapter 5: Fault Injection**

9 - **Chapter 6: Conclusions**

TeamName:
Daniel Hwang
David Rust
Kyle Stewart

Introduction

Our group, aka: TeamName, performed testing on NetHack - which is a singleplayer Roguelike video game released in 1987, featuring ASCII graphics. Upon beginning the testing process for NetHack, a project with no included tests we quickly became overwhelmed with the size and interconnectedness of the project. We had no idea where to start. What do you test in a large, graphical piece of software with no tests to begin with?

A TeamName group member contacted one of the maintaining developers to find out just that. Shortly after, we got a response from said developer explaining that NetHack had no tests, in part, due to the complexity and random components of the project. However, he pointed our group to some of the simpler-to-test string-parsing functions related to the wish parser - based on interpreting wishes from the player character in-game.

After our testing framework was set up, we chose to focus specifically on testing the two functions makeplural(char* string) and makesingular(char* string) in *objnam.c* for our eventual 25 total test cases. Once that testing functionality was added, several faults were injected into a separate version of the program to ensure the correctness of our testing process.

This document will display how to build and test NetHack using our testing framework and other guidelines in the repository at: https://github.com/csci-362-fall-2018-02/TeamName

TeamName:
Daniel Hwang
David Rust
Kyle Stewart

Chapter 1: Process to Compile Nethack for Ubuntu:

Nice resource: https://nethackwiki.com/wiki/Compiling

1. Clone from https://github.com/NetHack/NetHack.git. We'll say you save it in a folder named "Nethack".
2. Install dependencies. In the Terminal, run "sudo apt-get install bison flex libncurses5-dev" then type your password if any.
3. Instructions can be found in Nethack/sys/unix/NewInstall.unx. Gist of it is: there are "hints" files that automate the compilation process for certain systems, and in this case we can use the "linux" hints file.
4. Once you've followed the instructions in the NewInstall.unx file, navigate to your new NETHACKDIR (probably /usr/games/lib/nethackdir) and create a file called "sysconf" with the command "touch sysconf". Now you're compiled and installed! Run "nethack" in the terminal to play.

How did we get to this point?

Nethack is over 30 years old, and has had significant support from the developers. They have made it pretty straightforward for anyone to compile and install the game on numerous machines. By reading the README in the top directory from the github, it seemed like installing on Linux would be as easy as following the instructions in the "$top/sys/unix" folder, but just by looking in that folder, you can see there are actually something like 4 instruction files! After reading through these, the "NewInstall.unx" file was the newest, and seemed to simplify the process as long as our system isn't out-of-the-ordinary. We are using a fresh Ubuntu install, so just using the Linux hints file works fine. I had some trouble with the dependencies, but that was because I did not read all the resources available to me (specifically, the wiki page linked above is very clear about the dependencies). A bit of trial-and-error there had me compiled in short order. Running the game was more difficult, because I couldn't figure out where the "sysconf" file was supposed to go. Once again, that wiki page is very clear about this, so maybe the moral of the story is: read the wiki page on compiling, when that is an option to you! Lastly, there were no included test cases, and after discussion with the developers, their tests are not recorded in easy-to-distribute files.

TeamName:
Daniel Hwang
David Rust
Kyle Stewart

Chapter 2: "NetHack Test Plan" by TeamName

The Testing Process: First, we will focus testing on the Wish Parser and Object Naming functions found in objnam.c. Later tests may include areas such as Level Generation.

Tested Items:
objnam.c:

makesingular(*oldstr*) - Function receives a word as a string and returns the singular version of the word (e.g. "man" to "men," "sword" to "swords").

makeplural(*oldstr*) - Function receives a word as a string and returns the plural version of the word (e.g. "harpy" to "harpies," "dwarf" to "dwarves").

Testing Schedule: As of Oct. 19 2018

Deliverable 3 (Nov 9): By this date 5 test cases must be detailed and completed. For keeping with schedule in following Deliverables, 15 completed test cases would be preferred.

Deliverable 4 (Nov 19): By this date 25 test cases must be detailed and completed.

Deliverable 5 (Nov 28): Final report containing complete documentation of all test cases and procedures.

Test Recording Procedures:

All tests will include some kind of output, either to text files (preferred) or printed to the command line. Output will include timestamp records, scenario explanation, log of parameters passed and values returned, and result of the test: Pass, Fail, or another more detailed metric, if called for. Hardware and Software Requirements: Ubuntu 16.04 LTS Distribution Please follow the build instructions (see pg.1)

TeamName:
Daniel Hwang
David Rust
Kyle Stewart

Chapter 3: Testing Framework, Test Cases, and How-To

Testing Framework:

frame.py:

This is the driver of the framework. It takes input from the *exampleTests.txt* input file and sends it through *work.c* to interact with the NetHack source code. After the all of the values from the functions being tested are returned, *frame.py* also compares them with the expected outputs also given in *exampleTests.txt* and writes the results (including: PASS/FAIL data, time elapsed, expected output, and actual output) to the *testlog.txt* file.

work.c:

This is the C code that connects frame.py to NetHack source code and also contains some source code from NetHack itself that is necessary for ***makeplural()*** and ***makesingular()*** in ***objnam.c,*** the source code being tested, to be call-able.

junk.sh:

This script is needs to be run after building NetHack to properly link ***objnam.c*** with its dependencies (and its dependencies' dependencies, and *their* dependencies' depen- … *this goes on for some time*).

exampleTests.txt:

Our input file for what functions to test, what arguments to send, and what the expected output is.

testlog.txt:

Our output file for the given tests' results (including: PASS/FAIL data, time elapsed, expected output, and actual output).

Test-Cases:

Here are five test cases in the format in which they would be given as input.
(NOTE: only commands prefixed with a "$" identifier will be run in frame.py)

```
$ makeplural human humans
Base case: PASS assures basic consonant-s format sing-to-plur conversion.

$ makeplural valkyrie valkyries
Base case: PASS assures basic vowel('e')-s format sing-to-plur conversion.
```

```
$ makeplural man men
Badman: PASS assures basic -man to -men sing-to-plur conversion.

$ makeplural homunculus homunculi
-us to -i

$ makeplural larva larvae
-a to -ae
```

TeamName:
Daniel Hwang
David Rust
Kyle Stewart

Chapter 4: More Test Cases, Experiences

Our group now has 25 test cases (in *exampleTests.txt* in our GitHub repository) for the ***makeplural()*** and ***makesingular()*** functions in ***objnam.c***. All test cases are testing independent portions of the code, as they should, making them unique and non-redundant.

The framework is built and outputs: tested-output correctness, function being tested, it's arguments and expected output, as well as the total time elapsed on completion of the test.

TeamName:
Daniel Hwang
David Rust
Kyle Stewart

Chapter 5: Fault Injection

In the *objnam.c* file that we are testing adding faults was relatively simple. To do this we simply commented out sections of conditional statements that handled special word cases in the makeplural() and makesingular() functions we are testing.

In our particular case, we commented out the if-statement sections under the following comments in the program:

- algae, larvae, hyphae (another fungus part)
- man/men ("Wiped out all cavemen.")
- ium/ia (mycelia, baluchitheria)
- matzoh/matzot, possible food name
- Ends in z, x, s, ch, sh; add an "es"

After making these modifications we rebuilt NetHack, like so:

```
cd $Top/sys/unix      ($Top - NetHack main directory, the dir that contains the readme)
sh setup.sh hints/linux
cd ../..
make all
```

Add the test cases to exampleTests.txt in the framework folder:

```
makeplural larva      larvae
makeplural hypha      hyphae
makeplural amoeba     amoebae
makeplural man  men
makeplural matzoh matzot
makeplural mycelia  mycelium
makeplural balactheria  balactherium
makeplural mutex  mutexes
```

Run the frame.py script from your framework folder after first running junk.sh:

```
sh junk.sh
python3 frame.py
```

After running the script all of these cases should result in a FAIL result output.

TeamName:
Daniel Hwang
David Rust
Kyle Stewart

Chapter 6: Conclusions

Throughout this testing project we have dealt with a number of obstacles, whether it be: wrangling the project itself, group planning and communication, or being forced to make choices when building the framework.

Herding Cats:

Over the course of the semester group members have had differing availabilities, skill levels per area, and opinions about how to tackle the project. Each member learning to better navigate these differences has been a major takeaway from this project. Working in a group is often challenging, but crucial to the Software Engineering process.

The Right Tool:

Early on in the process of developing our testing framework we used exclusively the C Programming Language. Once it came to building functionality for string comparison and better file I/O, working only in C began to weigh on development time. This is when some of those tasks were re-assigned to Python to speed up writing the framework. Choosing the right tool for the job is important, and continuing with C would have taken longer for the same functional result.

A Nice Patina:

NetHack is quite an old project, and working with a large (to us) project like it was interesting. What seems to be a passion project doesn't always feature some of the stringent and more resilient design pattern or SWE practices that more modern or commercial software often tout. So in our troubles with NetHack were a few "what not to do" examples (not to criticize too harshly, NetHack is a really neat project all things considered).