Group 4: Isaac Supeene (isupeene) & Braeden Soetaert (bsoetaer)

Exercise 1: Shell

Questions:

Q: Is your problem a class or a module? What is the difference?

A: This problem is best illustrated as a class as the shell should be fairly self contained and shouldn't need to have its individual functions mixed in to other places. Also this allows instances of the shell to be created and allows the possibility of multiple instances.

Classes are typical OOP objects that contain methods and variables. They should be instantiated for most uses and mainly contain methods that are specific to the class and will not likely be used by other classes.

Modules on the other hand are more a collection of methods, constants and classes and should be mainly used to mix in methods that are used in multiple places into those places. They allow functions to be shared between classes without the classes having to know about the other classes or which specific class implements a useful method for what they are doing.

Q: What shell(s) are you using to provide a specification? What features do they support?

A: One of the most common shells, Bash, should provide us with a good specification of typical features that we should consider including in our shell. We will also look at Ruby's standard library “Shell” to see how they handle implementing some of the features.

Supported features:

- Change Directory
- Export
- Eval
- Exec
- Conditionals
- Looping
- Command/program execution
- Tab-Completion
  - Command
  - File path
  - File
- Automatic suggestions (Tab completion suggestions)
- Command History
- Directory history or stack
- Value prompt
- Menu/options prompt
- Mandatory argument prompt
- Implicit directory change
- Arithmetic
  - Integer Calculations without spawning external processes
  - Double parentheses arithmetic-evaluation
- I/O redirection
- Functions
- Exception handling
- Search and replace on variable substitutions
- Linear arrays or lists
- Array operations
- Associative Arrays

Eval function
Pseudorandom number generation
Shell scripts
Brace expansion (Alternation)
Startup scripts which require no execute permission or interpreter directive
- profile
- bash_logout
- bashrc

Command substitution using $() notation
String manipulation
Process substitution
REGEX
REGEX matching operator
Pattern matching (REGEX)
Pattern matching(filename globbing)
Recursive globbing (generating files from any level of subdirectories)
Coprocesses
Keyboard shortcuts
Batch vs concurrent execution
Execute process in background/foreground
Configurations
Logging
Pipes
Command substitution
Process substitution if system supports named pipes
Subshells
TCP/UDP connections as streams (Client only)
Secure password prompt
Execute permission

Q: In your opinion, which features are essential (should include in your design) and which are “window dressing” (should not include in your design)?
A:
<u>Essential</u>
Change Directory
Eval
Exec
Conditionals
Looping
Command/Program execution
I/O redirection
Functions
Exception Handling
Linear arrays or lists
Array operations
Shell scripts
Keyboard shortcuts (Especially control-C)
Batch vs concurrent execution
Execute process in background/foreground

Logging
Pipes

Q: Economics: Which, if any, essential features will be omitted from your design due to unmanageable effort requirements?
A:
Linear arrays or lists
Array Operations
Conditionals
Looping
I/O redirection
Functions
Shell scripts
Some less-needed keyboard shortcuts
Pipes

Q: Error handling? What percentage of code handles functional against potential pitfalls:
In the average commercial program? In your shell program?
If they are radically different, please provide a rationale.
A: It should have a higher percentage of error handling code than an average program because our program needs to be secure due to it being a system program. We do not want to expose parts of the system to the user if something goes wrong or an unexpected action/input is taken by the user.

Q: Robustness? How do we make the system bullet-proof? Is Avoiding Core dumps of system shells important? Especially from a Security viewpoint, remember this dump will give access to underlying C system code and potentially Linux daemons?
A: Make the system bullet-proof by validating inputs before executing them and limiting the user's powers in the shell. It is important to avoid core dumps as a core dump may contain important information that could jeopardize the security of the system inside them.

Q: Describe the Ruby exception hierarchy, which classes of exceptions are applicable to this problem?
A: The ruby exception hierarchy is ruby's set of built in exceptions/errors. Some of the classes of exceptions are applicable to this problem:
- SignalException – Interrupt for interrupting programs
- RuntimeError, SystemCallError
- NoMemoryError if running on machine with limited resources
- ArgumentError for invalid arguments

Q: What is Module Errno? Is it applicable to the problem? Explain your answer! Remember Ruby often wraps C code.
A: The Errno Module is a ruby module that dynamically maps the integer error codes from the underlying system to ruby classes where each class is a subclass of SystemCallError. Yes it can be useful when any C code is called because it can convert the integer errors into errors that make sense in Ruby and it can also convert these error classes back to the error numbers when calling C code. Many of the system programs are written in C code and being able to convert the error codes from these system calls into useful exceptions is good so that we can have useful errors.

Q: Security? How will we protect the system from tainted objects? Can we trust the user?
Is sand boxing applicable to this problem? Is it feasible to write security contracts?

A: We can protect the system from tainted objects by checking inputs in our shell and verifying the integrity of objects. Security is important in system level programming. No we cannot trust the user as you never know who the user might be. Sandboxing might be useful to isolate unknown programs from the rest of the system but this would be difficult to implement in the time constraint. The difficulty comes from the fact that many programs need access to other parts of the system and by containing them in a sandbox will often cause them to not function correctly. It is not feasible to write security contracts as there are an incredibly large number of ways that a system can be attacked by a malicious user and you will not be able to cover all attack vectors. This does not mean security should be ignored though but security contracts are very difficult to write and still won't cover all bases so they are not economically feasible to write.

Q: Should we be using class GetoptLong? Or Regexp? Or shell? Or ….
GetoptLong would be useful for getting the command line options for the different programs. Regexp would be useful if we decide to include regular expressions into our shell design. Shell would be useful to provide an example of how to implement certain functionality of our shell or we can compose with the class for some functionality.

Q: What environment does a shell run within? Current Directory? Or ….
It should run in the current shell or it could be started with an argument that tells it what directory to run within. The second option requires careful checking that the current user has access to that directory.

Q: What features should be user controllable? Prompts? Input and Output channels? Or ….
If prompts were included they should not be user controllable as the prompts could be modified to confuse users and ask for information that is not what is actually required. Input and output channels should be user controllable through input and output redirection but input should still be carefully validated.

Exercise 2: Timed Printout

Questions:
Q: A class or a module?
A: A module would be best for this problem as this should just be a small set of methods that can be included into other programs. The timed printout could be called from a single method.

Q: Error handling? Robustness? Security? Are any of these required?
A: None of these are really too stringent because this is a small program. Inputs should still be verified especially if a buffer of some sort is being used to avoid buffer overflow. We also have to be careful to not print out stack traces liberally when something goes wrong.

Q: What components of the Ruby exception hierarchy are applicable to this problem? Illustrate your answer.
A:
RuntimeError, SystemCallError for system calls that go wrong and runtime problems
NoMemoryError for systems with limited resources
ArgumentError for invalid arguments

Q: Is Module Errno useful in this problem? Illustrate your answer.
A: Yes because if we use C for this problem then it will be useful to know what error codes mean when

they come from C calls so that we can deal with them appropriately and correctly report them.

Q: Describe the article at:
http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html

Convince the marker that these Anti-patterns don’t exist in your solution.

Do they exist in your Shell solution?
A: This article describes patterns that are often seen when writing exception-handling and logging code that are bad patterns and should be avoided; Hence they are called anti-patterns. The anti patterns that exist and how we will avoid them in our timed printout:
Log and Throw
- Both logging an exception and throwing it which can cause clog in log files. Can be avoided by logging checked exceptions at the highest level that they are caught at.
Throwing Exception
- Throwing the basic exception instead of a specific one makes it so that users of your function just know that “Something could go wrong” and no specifics about what could go wrong. We can avoid this by throwing specific exceptions that correspond to the correct error.
Throwing the Kitchen Sink
- If there are multiple exceptions that are being thrown from a function we will wrap them in our own exception if all the exceptions should elicit the same response further up.
Catching Exception
- Catching the basic exception instead of a specific one causes changes underneath to go unnoticed even if a new specific exception is thrown that should be handled separately. We can avoid this by only catching specific checked exceptions and logging those and then we will notice changes below.
Destructive Wrapping
- This wraps an exception and destroys the useful information that was in the original exception. We will avoid this by maintaining exception info if we wrap it and hiding the exception info if we don't want to see it but not destroying the information.
Log and Return Null
- This logs the result so useful information is put to the log file but the information is lost inside the program. We will avoid this by only returning null when it is intended to be returned from a function.
Catch and Ignore
- This gets rid of the exception by catching it and doing nothing and provides no help in dealing with the problem. We will avoid this by at the very least logging exceptions when we catch them.
Throw from Within Finally
- This anti-pattern puts code that can potentially throw an exception in the finally clause or the ensure clause in ruby. We can avoid this by putting non-exception throwing code in the ensure block only.
Multi-Line Log Messages
- This anti-pattern puts logging info on multiple lines which causes problems when multiple instances are running. This could crop up in our solution but we will avoid this by putting useful logging info on a single line, potentially with an identifying factor.
Unsupported Operation Returning Null
- We can avoid this anti-pattern for our module by defining a method_missing for module functions that throws a NoMethodError. This is already done by the method_missing function that is implemented. Non-module functions it is more up to the class who includes our module.
Ignoring InterruptedException
- This anti-pattern is when interrupts are ignored by a program and don't immediately clean up what they are currently doing. We can avoid this by making our timer pay attention to interrupts so that the

timed printout will end the timer and print out a relevant message when it is canceled.

We will employ the same strategies to avoid these anti-patterns in our shell solution as we have listed above. We should pay more attention to the exception throwing and logging ones as they are more relevant to our shell solution than the timed printout. We may also need to handle the InterruptedException Anti-pattern differently in our shell and either pass the interrupt signal to the process that is currently being executed in the shell and not exitting the shell just because a SIGINT is received.

Q: How can I make the timing accurate? What time resolution should I be looking at, remember real-time systems? Time formats?
A: Can make the timing accurate by using C code such as nanosleep(). This allows us to have a time resolution in nanoseconds which is useful because it provides a far more accurate time resolution. The highest time resolution possible is necessary for real-time systems.

Q: Does ‘C’ have better facilities for this problem than Ruby? (Big hint!)
A: Yes C has better facilities for accurate time delays such as nanosleep. Ruby can not provide very accurate time delays for small time frames because the language is a relatively high level language.

Q: What should be user controllable? Can we trust the user?
A: The output message and the time delay should be user controllable. We should verify the time delay provided so that it is not abnormally large and the value provided is not larger than the system can support. The message should be relatively safe but if we store this message in a buffer we should ensure that this message will not cause buffer overflows.

Exercise 3: File Tracker

Questions:
Q: A class or a module?
A: This program is best represented as a module because the important part of this system is the methods that initiate the file tracking and these make it a good module.

Q: Error handling? Robustness? Security? Are any of these required?
A: Should handle errors where inputs of files to track are files that the user does not have permission to read or the file is located in a folder that the user does not have access to. For robustness we should handle tracking creation on files that are already created and for deleting files that don't exist.

Q: What components of the Ruby exception hierarchy are applicable to this problem? Illustrate your answer.
A:
SignalException (Interrupt) – If interrupt is received to stop file tracking.
ArgumentError – Error reading in arguments
IOError – Error if file can not be accessed or directory to monitor for file creation can not be accessed
RuntimeError – Exception if something goes wrong during run time
SystemCallError – Exception for when something goes wrong with a system call.
Errno::*

Q: Does this problem require an iterator?
A: While an iterator is one option, we could instead use the linux kernel's inotify subsystem to avoid

the need for an iterator to check for file changes. We would still need to iterate over the list of files when the functions are first called to set up inotify for those files.

Q: Describe Java's anonymous inner classes.
A: Anonymous inner classes are like local classes that have no name and are instantiated and declared at the same time. They can be used when you are creating a class that will only be used once.

Q: Compare and Contrast Java's anonymous inner classes and Ruby Proc objects; which do you think is better?
A: Anonymous inner-classes can contain more than one method but they need to implement an existing interface or extend an existing class. Ruby's Proc objects however are only 1 method each but they do not need to implement an interface or extend an existing class. They both allow for the defining of methods in the midst of another method. Also they both allow for the passing of methods to other functions as objects. We think that Ruby's Proc objects are better because they allow for simple definition of methods that can be passed around and called with arguments. We think they are superior to Java's anonymous classes because the main advantage you gain from Java's anonymous classes is that you can have multiple methods defined inside the class but if you are defining multiple methods for a class that is defined inline, you should probably ask yourself if this should be a real class instead.

Q: From a cohesion viewpoint, which interface protocol is superior? Explain your decision!
A: We believe that Ruby's Proc objects are better for cohesion because functions are defined in the procs and those functions typically belong together with the class that creates them. With Java anonymous inner classes though, entire classes are being defined inside other classes and if these classes are in fact different implementations of the same interface, these classes may not truly belong together because they do different things.

Q: Is Module Errno useful in this problem? Illustrate your answer.
A: Yes it is useful because we will be making system calls to check the status of the files and to set up monitoring on them. If any of these calls returns an error code we would need to convert this to the appropriate error class based on the Errno Module.

Q: Do any of the Anti-patterns described at:
http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html

Exist in your solution.
A: No we will return exceptions instead of null if a method is missing because we are not overwiting the method_missing function that does this already. The others we will code around and we will pay attention to the interrupt to stop file tracking.

Q: Describe the content of the library at:
http://c2.com/cgi/wiki?ExceptionPatterns

Which are applicable to this problem? Illustrate your answer.

Which are applicable to the previous two problems? Illustrate your answer.
A: The library in the link is a library describing common patterns of exceptions and good practices to use when using exceptions. Exceptions that are applicable to this problem:
NameTheProblemNotTheThrower – Provides useful exception names.
DesigningWithExceptions – Determine when to or not to throw exceptions.

BouncerPattern – Check arguments before calling module methods.
DontThrowGenericExceptions – Helps debugging as seen in anti-pattern.
DesignByContract – This is how we design our projects.
UseAssertions

Previous problems:

Shell:

NameTheProblemNotTheThrower - Provides useful exception names
DesigningWithExceptions - Determine when to or not to throw exceptions.
SecurityDoorPattern – Check permissions first for public door. Private door if root.
TidyUpBeforeThrowing – Tidy up running state from last command
ExceptionLogging – Log certain exceptions for information
DontThrowGenericExceptions – Helps debugging as seen in anti-pattern.
DesignByContract - This is how we design our projects.
UseAssertions – In our contracts

Timer:

DesigningWithExceptions - Determine when to or not to throw exceptions.
BouncerPattern - Check arguments before calling module methods.
DontThrowGenericExceptions – Helps debugging as seen in anti-pattern.
DesignByContract - This is how we design our projects.
UseAssertions – In our contracts

Q: Is a directory, a file? Is a pipe, a file? Is a ….., a file? Tell us your thoughts on the definition of a file in a LINUX context.
A: In Linux, everything is a file so we should be able to track most anything.

Q: Define what is meant (in a LINUX environment) by file change? Does it mean only contents? Or does it include meta-information? What is meta-information for a file?
A: A file change is any change to the files contents or its meta-information. This provides a better idea of when files are changed as everything is a file but everything may not necessarily have contents. Meta-information is permissions, timestamps, extended attributes, UID, GID, etc