Shell Program Questions

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

The difference between a class and a module is that a module is like a library. It cannot be instantiated, and serves to add in (mixin) to an existing object to provide extra functions.

Classes are only there to create objects. They can be instantiated, and cannot be included into something else.

Because our problem asks us to write a resuable basic shell utility therefore we are dealing with a module problem.

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

We will be taking inspiration mainly from the Bourne Again Shell (bash), and features from the Z Shell.

Q: In your opinion, which features are essential (should include in your design) and which are "window dressing" (should not include in your design)?

Essential Features:

- Privilege checking (sudo)
- Changing directory
- File operations
- Pipes

Window Dressing Features:

- Shortcuts (TAB completion)
- Command history (up/down arrows)
- alias/unalias
- bind
- echo
- printf
- Scripting

Economics: Which, if any, essential features will be omitted from your design due to unmanageable effort requirements?

- All of security (We will try to sanitize input to the best of our ability)
- Pipes

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.

Statistics for the average % of code in commercial program devoted to error handling were not found.

Our best guess for this value was around 10%.

The percentage of code that handles potential pitfalls in our shell program should be immensely high (initial estimate of 80%).

This is due to the shell program's nature, whose purpose is to accept any type of text input from the user,

and try to parse it into a command, and receive the output of that command, and display it to the user.

Thus, it has to be able to handle all the errors of the input along with all the potential errors of the output of the external programs.

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?

To make the system absolutely bullet-proof, we will handle all errors from all inputs and all outputs of the commands called. Because core dumps occur when bugs occur, and the bugs can be potential exploits, avoiding core dumps are important.

Describe the Ruby exception hierarchy, which classes of exceptions are applicable to this problem?

Ruby organizes its exceptions in the following manner:

```
Exception
  NoMemoryError
  ScriptError
    LoadError
    NotImplementedError
    SyntaxError
  SignalException
    Interrupt
  StandardError
    ArgumentError
    IOError
      EOFError
    IndexError
    LocalJumpError
    NameError
      NoMethodError
```

RangeError
FloatDomainError
RegexpError
RuntimeError
SecurityError
SystemCallError
SystemStackError
ThreadError
TypeError
ZeroDivisionError
SystemExit
fatal

Exceptions that are applicable to this problem are:

- RuntimeError
- SecurityError
- ThreadError
- SystemCallError
- SystemStackError
- IOError
- EOFError
- ArgumentError

What is Module Errno? Is it applicable to the problem? Explain your answer! Remember Ruby often wraps C code.

Module Errno maps operating system errors to Ruby classes, with each error number generating its own subclass of SystemCallError.

It is applicable to our problem by helping us read and treat errors from the operating system into the Ruby context.

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?

We definitely cannot trust the user to provide proper input. We will parse each input to make sure it is valid, before executing any command.
Sandbox Linux environments are applicable to test the security of this program.
It is not feasable to write security contracts because security involves niche cases that cannot be covered by pre and post conditions.

Should we be using class GetoptLong? Or Regexp? Or shell? Or ....

Classes GetoptLong and Shell are definitely useable.

The Shell class implements common UNIX shell commands,
while the GetoptLong class allows us to parse arguments in a UNIX shell fashion.

What environment does a shell run within? Current Directory? Or ....

The shell runs within the current directory.

What features should be user controllable? Prompts? Input and Output channels? Or ....

Aside from the user deciding how to craft their input, the user should not be able to control anything else.