1. We used a module to implement our shell. A module is essentially a library which provides various methods. A class is something which will be needed to create instances, having its own data and methods. The shell needs only to run, so different instances are not needed.

We used bash as a model to obtain our specification and guideline. It supports a wide variety of features, including file and directory manipulation, scripting capabilities, operation piping, process creation and handling, and looping functionality.

We decided that for our purposes, the file and directory manipulation were the most important, and the other functionality while useful, is not essential for all users.

Our shell does not support any process handling or scripting. It also can not time processes, nor does it run any compilers. It does not offer tab completion or command memory.

Since the shell will be running on the user's system, a core dump will only potentially expose their own registers and daemons. It will not provide them with any sensitive information.

This is a list of the entire ruby exception hierarchy: (Excluding ERRNO values, which all fall under SystemCallError)

```
BasicObject
  Exception
    MonitorMixin::ConditionVariable::Timeout
    NoMemoryError
    ScriptError
      LoadError
        Gem::LoadError
      NotImplementedError
      SyntaxError
    SecurityError
    SignalException
      Interrupt
    StandardError
      ArgumentError
        Gem::Requirement::BadRequirementError
      EncodingError
        Encoding::CompatibilityError
        Encoding::ConverterNotFoundError
        Encoding::InvalidByteSequenceError
        Encoding::UndefinedConversionError
      FiberError
      IOError
        EOFError
```

```
  IndexError
    KeyError
    StopIteration
  LocalJumpError
  Math::DomainError
  NameError
    NoMethodError
  RangeError
    FloatDomainError
  RegexpError
  RuntimeError
    Gem::Exception
      Gem::CommandLineError
      Gem::DependencyError
      Gem::DependencyRemovalException
      Gem::DependencyResolutionError
      Gem::DocumentError
      Gem::EndOfYAMLException
      Gem::FilePermissionError
      Gem::FormatException
      Gem::GemNotFoundException
        Gem::SpecificGemNotFoundException
      Gem::GemNotInHomeException
      Gem::ImpossibleDependenciesError
      Gem::InstallError
      Gem::InvalidSpecificationException
      Gem::OperationNotSupportedError
      Gem::RemoteError
      Gem::RemoteInstallationCancelled
      Gem::RemoteInstallationSkipped
      Gem::RemoteSourceException
      Gem::RubyVersionMismatch
      Gem::UnsatisfiableDependencyError
      Gem::VerificationError
  SystemCallError
  ThreadError
  TypeError
  ZeroDivisionError
SystemExit
  Gem::SystemExitException
SystemStackError
fatal
```

ERRNO provides wrappers around operating system specific errors to insert them into the Exception hierarchy in ruby. This is useful when wrapping C code, because the errors caught will not necessarily correspond to ruby errors. Since we will be using Swig to wrap some functions, we will likely be able to make use of ERRNO to catch various errors.

The system only accepts inputs which it knows and trusts, so users will not be able to run destructive commands.

Sandboxing is feasible for this application, since or shell will simply be a small shell sitting on top of the existing bash. Our application will limit the user's capabilities in order to maintain security.

We will be using class GetOptLong to accept and parse user input. The shell runs in the user's system space, so all of their accessible files and directories are potential environments. The user can control where output from their commands goes, but that it it.

2. The delaying functionality will be implemented in a C class which will be converted to a ruby module using swig. The driver is a Ruby class and it handles the functionality of taking the command line arguments and pass them into the delay function before creating a background process that prints the specified message after the specified delay.

Since this is just a single command with two parameters, the only component of the Ruby exception hierarchy useful would be ArgumentError. As the user has a very limited choice of input and since we deal with invalid input using exception handling, there would not be a security concern. Module Errno is used to take the errors reported by the operating system in the form of integers and create corresponding Ruby subclasses in the Ruby exception hierarchy. This module will be used in the driver to deal with the invalid arguments exceptions.

The article describes various exception handling anti-patterns in Java. These anti-patterns include: logging as well as throwing the exception, throwing just a general exception without declaring it, throwing multiple exceptions that mean the same thing, catching a general exception, wrapping an exception in another exception which destroys the stack trace of the original exception, returning null after logging, simply ignoring the exception and returning null, and ignoring an interrupted exception. Even though these anti-patterns are described for java they can be translated to apply to ruby. Our exception handling does not include these anti-patterns since we have not implemented the actual design yet.

To make the timing accurate, it would be useful to use milliseconds for the time resolution. In real-time systems any delays caused by lack in granularity of the time resolution used would be unacceptable. Since Ruby is not suited for real-time applications and does not provide high precision timing functionality, we will use C for the implementation.

The only thing controllable by the user is the delay and the message to be printed. Since the user provides these as command line arguments and any invalid inputs are handled, we can trust the user.

3. This will again be a module, as only methods are required, not instances.

This will not require much security, as the function only watches for changes in a file, and then executes a user specified command, both of which are safe operations.

SystemCallError and ArgumentError will be applicable to this part. The former contains ERRNO, and will handles any errors encountered in the C programming, and the latter will be used for errors in the execution of the functions, such as attempting to run an illegal command.

An iterator could potentially be used to iterate over multiple commands in the user specified block.

Java's anonymous inner classes can be used to essentially create a class in place inside another class. This can be helpful in passing small classes as parameters to functions, or for implementing things that need to access local variables but require a degree of separation.

Proc objects can be assigned to a variable, and can be reused anywhere with any inputs. Multiple procs with different settings can be bound to different variables to make similar functions. Ruby's Proc objects seem to be more versatile and applicable to more situations.

I suppose the single interface protocol exhibits a higher degree of cohesion. It can be extended to add more types of watchers, and it bundles all of the related functions together into one.

None of the anti-patterns exist in our solution currently.

The article describes best practices for generating, naming, and handling exceptions. Something like Catch What You Can Handle could be applicable to our solution

In Linux, essentially everything that is not a process is considered a file by the system.

Any change to the file should be considered an alteration. Meta-information for a file includes information about that file, such as size, access times, and protection.