Software Engineering 200: Mars rover assignment

Delan Azabani

October 21, 2014

To compile and run this assignment submission, use the following commands:

```
% ant && java -jar dist/rover.jar
```

1 Part (c): system walkthrough

Let's say that mission control sent the following message to our rover on Mars:

```
analyse;translate:-100;photograph;rotate:-180
```

Each message represents a task list, where the tasks are separated by semicolons. Each task has a name, such as one shown above or `call`, which executes a previous task list. Parameters for a task such as angles, distances and task list numbers are written after their task name, separated by a colon.

From the perspective of our rover's system, this message starts at `Comm.receive(String)`, which is implemented by `ConcreteComm.receive(String)`. Because `ConcreteComm` is a subject in the Observer pattern, all `receive()` needs to do is call the subject's method named `notifyObservers()`, which required by the `ObservableDevice` interface.

`notifyObservers()` then iterates through each of the `DeviceObserver` objects who are listening, calling their `update()` methods with the message string from `receive()`. In practice, the only observer of `ConcreteComm` is one `CommObserver` object.

`CommObserver.update()` takes the message and simply downcasts it to a `String`. Then it finishes by passing it to `controller.messageReceived()`, where `controller` is the `RoverController` that owns the `CommObserver`.

`RoverController.messageReceived()` instantiates a `RoverCommandCodec` to parse the message and turn it into a task list. There's an unfortunate level of coupling here because the task objects that its `decode()` method creates need to have references to various devices (a `Camera`, a `Driver` and a `SoilAnalyser`). Worse still, a `RecursionCommand` needs to be able to read existing task lists and modify the execution stack. As a result, the `RoverController` needs to pass many parameters to `decode()`.

In return, we now have a `RoverTaskList` containing a `RoverCommand` for each task in the incoming message, and we append this task list to `program`, a `RoverListList` which represents a list of task lists. What a mouthful.

In another thread, `RoverController.start()` has been polling an execution stack called `stack` jadedly, because thus far it has remained empty. The stack simply contains `Iterator` objects inside active task lists. As there's now a task list in `program`, we take an iterator for it and push it onto the stack. Whenever `busy` is `false` (iff no tasks are running) and there's a task list on the stack, we grab the next task with `Iterator.next()` and call its `execute()` method.

`AnalysisCommand.execute()` calls `soilAnalyser.analyse()` using its private reference field, and `ConcreteSoilAnalyser` sends the results of the analysis back to the controller using the same Observer workflow discussed previously.

Finally, `RoverController.analysisReceived()` calls `Comm.send()` and Earth is pleased with the rover. Rinse and repeat with the rest of the tasks and task lists.

2 Part (d): design patterns

The Observer pattern is essential and works well with the asynchronous nature of this system. It's significantly cleaner than setting and polling for changes in global state. `Concrete{Comm,Driver,SoilAnalyser,Camera}` are the subjects, each implementing the `ObservableDevice` interface, while `{Comm,Driver,SoilAnalyser,Camera}Observer` are the observers, implementing `DeviceObserver`. `RoverController` can even be seen as an overarching observer of the other observers.

Dependency injection is used to decouple the `RoverController` from any particular set of concrete device classes that it uses. `RoverProgram` is the entry point of execution, and it is responsible for injecting instances of `Concrete{Comm,Driver,SoilAnalyser,Camera}` or any subclasses of these into a new controller object.

The Factory pattern is realised through `RoverCommandCodec`. By refactoring task list message parsing out of `RoverController`, the code which instantiates various subclasses of `RoverCommand` is isolated in its own class without any other functionality.

`RoverCommand` is an interface that demonstrates the Command pattern in combination with `{Rotation,Translation,Analysis,Photography,Recursion}Command`, the classes that implement it. The key benefit of using this pattern here is that it allows the code in `RoverController` to deal purely with coordinating the execution stack, while treating tasks as opaque entities. All the controller does is choose the time of execution.

The container classes `RoverListList` and `RoverTaskList` are subjects of the Iterator pattern, as it allows the `RoverController` to keep track of a 'cursor' in each task list without the need for clumsy integral indices. Actually I ended up being unable to take advantage of the pattern with `RoverListList`, because the iterator would die whenever the list of task lists is modified, which occurs whenever a new task list arrives.

3 Part (e): alternative design choices

With my final design, I couldn't take advantage of Java's native implementation of the Observer pattern, because `Observable` is a class, not an interface, and I was already using my implementation inheritance relationship with `Comm`, `Driver` and the like. I could have instead decorated `Concrete*` with four classes that extended `Observable`. While this would obviate the need for boring Observer code, it would introduce delegate methods instead. I chose the route with fewer classes overall.

`RecursionCommand` is unlike the other task classes, because it doesn't really *do* anything other than push an old task list reference onto the execution stack. There's no device associated with it, and thus no observer to tell the controller to reset `busy`. I ended up working around this by 'tagging' the other task classes with the empty interface `AsynchronousCommand`, and checking for this 'tag' with `instanceof` after firing off `execute()`. One could argue that this may be an anti-pattern, as it's a bit of a semantic abuse of the interface language feature. After finishing the assignment, I discovered that this technique was common before Java introduced annotations; had I known about annotations earlier, I would have certainly preferred to use them instead.

4 Part (f): separation of concerns

At the risk of rehashing my answer to part (d), virtually every design pattern I used was chosen at least partially due to its capacity to improve the separation of concerns within classes inside the Mars rover system.

The Observer pattern separated what was to happen after a device had new information, from each device's specific events and their implementations.

The Factory pattern separated the choice of concrete task classes from `RoverController`, while dependency injection separated the choice of device classes from its constructor.

The Command pattern separates the 'when' from the 'how' of executing each task.

The Iterator pattern separates the act of walking through tasks from the underlying data structure used, because only the former is relevant to the controller class.

5 Part (g): avoiding unnecessary duplication

The design used in this assignment does not use any delegate methods or decoration classes outside of `RoverListList` and `RoverTaskList`, which are basically encapsulating *semantic* sugar around what are essentially `ArrayList` objects.

I've taken advantage of the existing relevant exception class `IllegalArgumentException` rather than creating my own subclass just for this system. For readers of the source code, this provides the slight additional benefit of familiarity.

The use of tedious `switch`-like constructs on external input has been restricted to one such usage inside `RoverCommandCodec.decode()` only.

The frequently used `Thread.sleep` wrapped in a `try-catch` for `InterruptedException` has been refactored into a static method in `RoverUtils` for clarity of reading.

6 Part (h): testability

As the name suggests, the package `com.azabani.java.rover.fake` contains fake versions of the classes `Comm`, `Driver`, `SoilAnalyser` and `Camera`. While these already existed hypothetically, and are external to the system, they were important for four reasons:

- They allowed me to build the system during development and check the syntax;
- The compiler would also check data types, a weak form of semantic verification;
- I could craft a limited set of test data to use as an occasional sanity check; and
- Being able to run parts of the system while writing it allowed me to better visualise where I was going with the system, because I felt less 'blind'.

Given more time, I would have developed unit test harnesses using JUnit and Mockito to test the system in a much more rigorous manner, employing the techniques covered in *Software Engineering 110*. The use of the Factory pattern and dependency injection would certainly help with unit testing, because of the way they effectively decouple classes.