CURTIN UNIVERSITY

SE200 Assignment Discussion

Alex Burress

1457 1008

Design and Implementation

Step-by-Step Walkthrough

For this example, I'm assuming the main algorithm is running an infinite loop with the RoverController and Rover already instantiated. Let's say the rover receives the String "875; D 45.0;T -13.5;A;P;L 345". This String is asking the rover to perform the following tasks in order:

1. Drive 45.0 metres forward.
2. Turn 13.5 degrees anticlockwise.
3. Perform a soil analysis.
4. Take a photo.
5. Perform the List of tasks previously sent with the identifier 345.

The number 875 at the start of the string is the identifying number for this list of tasks.

The message is received by the Rover through the receive() method in the RoverController class. The message is split using the delimiter ";" and stored in the splitMessage String array. After being split, splitMessage[] contains the following Strings:

1. 875
2. D 45.0
3. T -13.5
4. A
5. P
6. L 345

The splitMessage[] array is used to create a TaskList object. The TaskList constructor takes in the String array and the identifying number and converts each String except for the first in splitMessage into a Task using a for-each loop and a factory. The createTask() method in TaskFactory is called for every String after the first in splitMessage[]. createTask() looks at the first character of each String and calls the relevant Task constructor. In our example the following occurs:

1.	875	->	Nothing, string is skipped
2.	D 45.0	->	A Drive Task is created
3.	T -13.5	->	A Turn Task is created
4.	A	->	A ConcreteSoilAnalysis Task is created
5.	P	->	A TakePhoto Task is created
6.	L 345	->	The task list corresponding to the identifier 345 is retrieved

These tasks, and the identifier are stored in a TaskList object called newList, which represents a series of tasks to perform in a certain order. newList is then added to a queue of other task lists awaiting execution via the rover.addToBacklog() method call. In this example, we will assume that the backlog was previously empty, which means our newly added task list will be at the front of the queue.

A call to doFirstJob() in the RoverController class is now made. The first TaskList in the backlog is removed and has its executeTaskList() method called. executeTaskList()iterates through each task in the list and calls the performTask() strategy method. In our example performTask() is called on the Drive Task, this in turn calls the drive() method in the Driver superclass. It is assumed that the drive() method eventually calls either moveFinished() or mechanicalError(). In this example we assume that moveFinished() is called and the RoverController is notified that the rover successfully drove forward 45 metres and the next iteration in executeTaskList() can begin. The next three iterations in executeTaskList() call performTask() on the Turn, ConcreteSoilAnalysis and TakePhoto tasks. Each of these Tasks notify the observing RoverController that their tasks were successfully performed. The final iteration in executeTaskList() calls performTask() on the TaskList corresponding to the identifier 345. performTaskList() calls executeTaskList() for that particular TaskList and calls performTask() on each task in that TaskList, in this example we assume that TaskList 345 only has one Drive Task. Once the Drive Task has completed and notified the RoverController, RoverController is then notified that List 345 has completed successfully and TaskList 345 is re-added to the finished jobs map in RoverController's Rover object. Next RoverController is notified that TaskList 875 has completed successfully and the TaskList is added to the finished jobs map as well.

Discussion

Design Patterns Used

Strategy pattern

The Subject and Task interfaces allow for encapsulated Tasks with the ability to perform themselves and notify the controlling observer when an event occurs with common method calls. I used this pattern because it allows for different algorithms for task execution to be chosen at runtime.

Iterator pattern

The iterator pattern is used twice in the TaskList class for iterating through containers. The iterator pattern keeps a layer of abstraction between my algorithms and the inner workings of the containers I use.

Observer pattern

This pattern is used between Tasks and the RoverController. Tasks are subjects and RoverController is an observer. This pattern is used to free up the controller whilst tasks are performed in the background. The observer pattern also creates a layer of abstraction between Tasks and the RoverController, neither are directly coupled with concrete observer or subject classes respectively.

Composite pattern

A composite pattern is used between TaskList (branch) and the Drive, Turn, TakePhoto and ConcreteSoilAnalysis classes (leaves). All of the aforementioned classes implement the Task interface. This pattern was chosen to facilitate recursive execution of TaskLists containing other TaskLists.

Factory pattern

TaskFactory.createTask() analyses the passed in string and instantiates the relevant type of Task object. This pattern reduces coupling between Tasks and TaskLists and ensures that the TaskList class does not need to be aware of each type of concrete Task class.

Plausible Alternative Design Choices

Decorator pattern

I could have had tasks inherit from Observer and used the decorator pattern to construct chains of Tasks to perform. In this configuration each inner Task would have notified its outer Task when it was finished so the outer Task could start, this would continue until the outermost Task completed or an exception was thrown. If the outermost task completed successfully it would have informed the RoverController, if an exception was thrown the Task that caught the exception would have notified the RoverController which would have then aborted the task chain.

I chose not to implement this pattern because I believed it would be very difficult to implement, and would make my code harder to understand and therefore maintain. I also believed that a decorator pattern as described above would have made my system harder to test than my current design due to increased coupling between tasks.

Singleton pattern

My system only uses one instance of a Rover object, which is in turn accessed by multiple classes. I could have safely implemented a singleton for this design as it stands, however, I felt that not implementing rover as a singleton gave the system more flexibility for future changes. What would happen if NASA sent out a second rover? In this scenario a singleton would necessitate more code modification and testing than my current design.

Separation of Concerns

My system achieves separation of concerns with encapsulation, architectural separation and packages.

Encapsulation

Wherever possible I coded to interfaces rather than implementations. I used three interfaces, namely NasaObserver, Subject and Task. Communication to and from these classes went through interfaces, aiding encapsulation.

Although a trivial example, all of my class fields were declared as private, which added a layer of encapsulation to the state of my objects.

Architectural Separation and Packages

My system implemented a Model-Controller-Exception distinction between packages, no user interface existed in this system so I did not categorise any classes as part of the view. All of my custom exception types were in the same package. My model package only consisted of the Rover class, which held task data with accessors and mutators. The controller package was further split into factory and task sub-packages.

Architectural separation is well demonstrated by the Rover and RoverController classes. The Rover class only holds data whereas RoverController holds algorithms for business logic using Rover object data.

Upon reflection, I could have separated the algorithms in TaskList into two separate classes. The TaskList class stores data in a linked list and includes accessor methods for the stored data. It also has business logic in the constructor and executeList() method. If I were to refactor this code I would store TaskList in the model and shift the business logic into an ExecuteTaskList class.

Reuse

I believe that my use of the iterator, composite and factory patterns has reduced code reuse. I was able to use a for-each loop to iterate through a linked list and array, which saved me from having to code custom iterators that would have had similar algorithms to each other.

The composite pattern saved unnecessary code due to its recursive structure, the system can just keep executing TaskLists until there are no more branches left with a single method call.

The factory pattern put all Task instantiation in one place. The task factory is only called from one method in the whole system, however, if the system were added to and instantiation were needed in any other classes the factory pattern would definitely save code reuse.

Testability

To increase the testability of my system I used dependency injection wherever possible, when I invoked methods I passed in references to objects already on the stack rather than new objects, unless they were standard Java object types.

I also chose to not use the singleton pattern, which can make testing more difficult.

If I were to refactor my code, I would have the TaskFactory represent an object type rather than a class of static functions. Coding the createTask() function as static has created a hard-coded dependency between TaskList and TaskFactory. I would also modify some of my methods that throw Exceptions to instead return null if an error occurs, this would allow for more mocking and increase the scope of unit testing.