Rovers

Design and Development

# Overview

This document is a brief review of the design and development process for the *Rovers* project. Discussed is the motivation and goals, a review of the requirements, a project plan with rationale and the design of the final solution.

# Motivation

The Rovers project aims to develop a solution for the problem of exploring a plateau on Mars with a squad of robotic rovers. A detailed description can be found in *problem\_description.txt*. However, a second and arguably more important goal is to demonstrate Daniel’s proficiency as a professional software engineer. This includes demonstrating a disciplined decision making process, an understanding of the complete SDLC and utilization of effective software engineering practices and tools.

# Requirements

The problem description specifies most of the system adequately, thus in the interest of brevity only ambiguous requirements, assumptions and the resolutions are listed here. The requirements won’t be formally tested themselves, but will drive the development and unit tests.

1. The plateau is a finite, uniform 2d grid.
2. All grid coordinates, dimensions and rover (X, Y) positions are positive integers.
3. A rover's facing is given as N, E, S or W, matching North, East, South and West respectively.
4. A rover will halt and ignore future commands if the next command is invalid.
5. A rover command is invalid when it results in moving off the plateau edge, crashing into another rover or is not a character in the set {'L', 'M', 'R'}.
6. A spin command, {'L', 'R'}, is relative to the rover's facing.
7. A move command, 'M' moves in the direction of the rover's facing.
8. A rover perfectly receives all sent commands.
9. A rover’s actions are independent of other rovers.
10. A rover's position is always denoted by an X Y F tuple, matching x-coordinate, y-coordinate and facing respectively.
11. A rover that is executing spin commands is classed as 'moving', implying the next rover must wait.
12. A single (X, Y) position corresponds to a maximum of one rover. For multiple rovers, this implies they'll start and end in non-overlapping positions.
13. Multiple rovers starting in the same (X, Y) position aborts the simulation (physically impossible universe!).
14. Input and output commands are ASCII encoded
15. Successive rovers will being exploring immediately after the previous rover finishes.
16. Simulation input can be read from a file or standard input.
17. The file format always follows the problem\_description.txt example perfectly.

Requirement 4 is chosen to mimic and handle uncertainty in the environment – the robot will initially attempt to complete its mission, but if it encounters an impossible scenario, it’ll stop and await new commands. Requirement 5 is a reasonable choice to prevent rover losses. Requirement 16 makes bulk test runs easier.

# The Plan

To complete the solution successfully and to budget whilst also demonstrating good engineering practice encourages an agile approach. Although the process benefits are limited given the project’s scope and one-person development team, the value of CI and TDD alone make it ideal. Further, taking an iterative approach allows us to illustrate effective requirements analysis, task prioritization and change and risk management as demanded by larger projects.

To demonstrate these methods well, two thoughts drive the project:

* *Design minimally, but build for change.*
* *You ain’t gonna need it.*

By minimizing up front design whilst still meeting the requirements, we ensure the solution is always minimal but complete. Further, by encouraging change we can demonstrate how refactoring can be used to build a design over time.

The tasks are now prioritized as follows:

1. *A single rover can \_act\_ (move and rotate).*
2. *A rover can perform multiple successive actions.*
3. *A rover can be commmanded with ascii command strings.*
4. *A rover moves only to unoccupied squares.*
5. *A rover gracefully stops when an invalid command is received.*
6. *Multiple rovers can explore the plateau.*
7. *A user can input an environment (plateau, rovers) and rover command lists.*
8. *Successive rovers explore without collisions.*
9. *All user-input rovers output the final position of the exploration path.*
10. *A user receives the final positions of all user-input rovers*
11. *A user can receive help using the program.*

The tasks are prioritized in order of importance. Although an incomplete solution is adequate, developing against the priorities affords the *possibility* of releasing a functional -if incomplete- solution early. For a larger project such stories would be further broken down.

## Testing

There are three primary levels of testing – system, integration and unit. The solution applies unit-testing to all components, including objects and external functions. Integration tests are built for testing inter-component communication. Given the limited scope and time available, system testing will be driven with manual tests of the solution against pre-verified test data. This is sufficient given the degree of confidence afforded by unit plus integration tests.

## Documentation

Beyond the design document a journal is maintained tracing development. The source code is documented using doxygen compatible markup allowing automatic generation of API documentation. Markdown format is used for non .DOCX documents and be transformed to HTML. Finally, the git log can be captured for further project insight.

# The platform

To create an effective but light TDD, a combination of Git, CMake, Google Test and shell scripting is used. The framework is lightweight and allows the user to easily build the project and execute the unit and integration tests. The use of Git ensures a full project history is always locally accessible, while its performance affords rapid experimentation. CMake with the included bootstrap scripts allow the repository to change erratically yet be automatically scanned and rebuilt. Finally, Google Test is a strong and succinct unit-testing framework.

The solution is implemented using C++ TR1 and a combination of STL and Boost. Although the temptation to embrace C++11 is great, the risk of build failures across platforms was too high. The desire to use Boost is mixed. The Boost library provides an enormous range of high-performance and well-tested components still not implemented by C++11. However, the documentation is often lacking and the complexity of many libraries, both in technical understanding and error messages has made it unattractive for many developers. The author’s perspective is pragmatic – for an individual project, Boost contains many useful components. However, in a diverse team, it may be better not using Boost at all.

# the design

The design, although constrained to the agile plan proposed, is motivated by several aspects:

* Make the solution a single module, command-line application.
* Reuse existing tools and algorithms wherever possible
* Focus on simplicity – complexity is the biggest killer of software development
* Design for (unit) testing
* Separate the Rover API and rovers client

The overall theme is keeping the implementation as simple as possible without unnecessary over-engineering. This also implies using referentially transparent and generic programming over OOP where possible.

There are two main business objects that must be realized by the solution, *Rover* and *Plateau.* A *Rover* is responsible for moving itself and exploring a plateau. A *Plateau* represents the plateau being explored by a squad of rovers and defines the rules of rover and plateau interactions – collisions, falling off the plateau, etc. An *Application* concept class exists to manage the control logic and Rover and Plateau models.

The associations can be represented with the following very simple class diagram:



*In the interest of brevity and the awkwardness of the online UML tools I found I’ll omit an interaction diagram, I really gotta wrap this up anyway!*

The primary responsibilities of the Rover API is the *simulate\_squad* operation, allowing a squad of Rovers to explore a Plateau given an existing list of commands.

# Fetchez la vache!

Not typically a section in a design document, but I just wanted to say thanks for reading, I hope you enjoyed it as much as I did writing it. After all, I think being casual when appropriate is an essential ‘spice’ as an engineer.