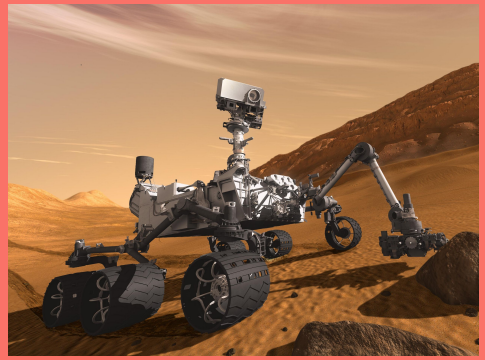
Design Patterns

- Exercise
- What are they… And why?
- Advantages and pitfalls
- Exercise

Let’s start with an exercise

Mars Rover

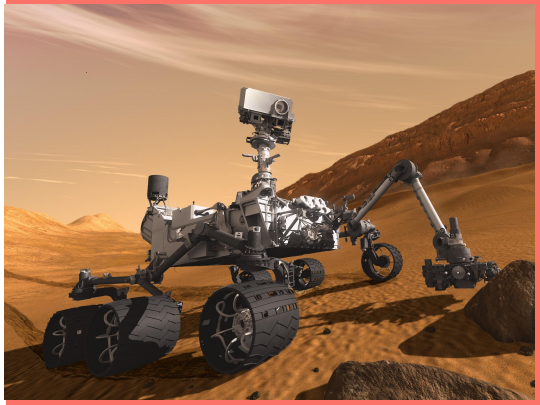
- A squad of robotic rovers are to be landed by NASA on a plateau on Mars.
  This plateau, which is curiously rectangular, must be navigated by the rovers so that their onboard cameras can get a complete view of the surrounding terrain to send back to Earth.
  Your task is to develop an API that moves the rovers around on the plateau.
  In this API, the plateau is represented as a 10x10 grid, and a rover has state consisting of two parts:
  - its position on the grid (represented by an X,Y coordinate pair)
  - the compass direction it's facing (represented by a letter, one of N, S, E, W)
- The input to the program is a string of one-character move commands
  - L and R rotate the direction the rover is facing
  - M moves the rover one grid square forward in the direction it is currently facing
- If a rover reaches the end of the plateau, it wraps around the end of the grid.
- The program's output is the final position of the rover after all the move commands have been executed. The position is represented as a coordinate pair and a direction, joined by colons to form a string. For example: a rover whose position is 2:3:W is at square (2,3), facing west.



Let’s start with an exercise

Mars Rover

- The grid may have obstacles. If a given sequence of commands encounters an obstacle, the rover moves up to the last possible point and reports the obstacle by prefixing O: to the position string that it returns. For instance, O:1:1:N would mean that the rover encountered an obstacle at position (1, 2).
- Example inputs
  - given a grid with no obstacles, input MMRMMLM gives output 2:3:N
  - given a grid with no obstacles, input MMMMMMMMMM gives output 0:0:N (due to wrap-around)
  - given a grid with an obstacle at (0, 3), input MMMM gives output O:0:2:N



At this moment, it should be a bit of a known topic, but anyway, we'll make a quick refresh. For starters, we acknowledge their usefulness on how to tackle certain coding tasks.

The problem is that, we've seen many examples of over engineered software driven by applying Design Patterns "by the book"

Design patterns: Advantages

- They give you a range of scenarios of recurring problems already solved in which to look for inspiration.
- They give you a vocabulary for talking about possible implementations of a solution, so that there is no need to go into detail.
- They give you a quick reference for a possible implementation of a solution if the problem fits.
- The implementations used are based on fundamental software design principles that have been widely accepted.

Design patterns: Pitfalls

- They almost never apply out-of-the-box, but need to be shaped and customized to match a particular scenario.
- They fail miserably to express business concepts and domain vocabulary, focusing on implementation details instead of behavior.

Note: Avoid leaking the name of the pattern in your code when possible. Naming should always focus on behavior related to the business functionality.

Design patterns: Creational patterns

Problem	Solution	Pattern
Are we instantiating families (or sets) of objects?	Provide an interface for instantiating families of related or dependent objects without having to specify their concrete classes	Abstract Factory
Do derived classes need to create instances following some logic?	Define the interface for object creation, but delegate decisions about instantiation to the subclasses	Factory Method
Do we need to follow several steps to instantiate our objects?'	Separate construction of an object from its representation so that the same construction process can create different representations.	Builder

Design patterns: Behavioral patterns

Problem	Solution	Pattern
Do we apply some rules or algorithms that are replaceable or can change?	Define families of algorithms, encapsulate each one behind an interface, and make them interchangeable	Strategy
Do we have an algorithm that needs to change certain behavior in subclasses?	Define the abstract steps of a superclass, deferring some steps to subclasses	Template Method
Do we need call back functionality or to keep track of history? Do requests need to be handled at different times or in different orders? Should we decouple the invoker from the object handling the invocation?	Encapsulate the method call in a first class object. This allows the request to be handled with traditional constructs such as lists, queues and callbacks	Command

Design patterns: Behavioral patterns

Problem	Solution	Pattern
Do several different entities need to know about events that have occurred?	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified. Create an event-based communication initiated by the dependent object going opposite the dependency flow	Observer
Do we have a system with lots of states where the logic for the different states is complicated?	Allow an object to alter its behavior when its internal state changes. The object will appear as if it changes its class.	State
Do we have different objects that might do the job, but we don't want the client object knowing which is actually going to do it?	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it	Chain of Responsibility

Design patterns: Behavioral patterns

Problem	Solution	Pattern
Do we have new tasks to be applied to our existing classes?	Represent an operation to be performed on the elements of a structure	Visitor
Do we want to separate a collection from the client that uses it so we don't have to worry about the collection's implementation details?	Provide a way to access the elements of the collection sequentially without exposing its underlying representation	Iterator

Design patterns: Behavioral patterns

Problem	Solution	Pattern
Do we have a lot of Coupling in a many-to-many relationship of object interactions?	Define an object that encapsulates how a set of objects interact, promoting loose Coupling by keeping objects from referring to each other explicitly, letting their interaction vary independently.	Mediator
Do we have an internal state of an object that must be saved and restored at a later time and cannot be exposed by interfaces without exposing implementation? Must encapsulation boundaries be preserved?	Without violating encapsulation capture and externalize an object's internal state.	Memento

Design patterns: Structural patterns

Problem	Solution	Pattern
Do we need to add some optional new functionality to something that already exists?	Provide an abstraction or placeholder for another object to control access to it.	Proxy
Do we have the right logic exposed with the wrong interface?	Convert the interface of a class into another correct interface.	Adapter
Do we have multiple additional functions to apply, but order and cardinality is variable?	Add dynamic additional responsibilities to an object.	Decorator
Do we want to simplify, beautify and clean existing systems or sub-systems?	Provide a unified interface to a set of interfaces in a system.	Facade
Do we have several different objects that need to be logically aggregated in a whole?	Compose objects into tree structures to represent part-whole hierarchies	Composite

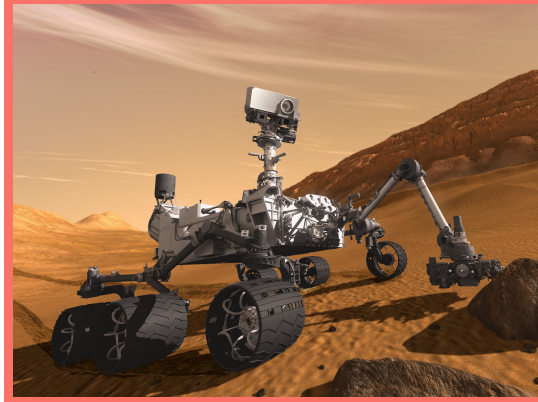
Revisiting refactoring guidelines

- Extract private methods from deep conditionals.
- Extract smaller private methods from long methods, and encapsulate cryptic code in private methods.
- Return from methods as soon as possible.
- Encapsulate where we find missing encapsulation.
- Remove duplication.
- **Refactor to patterns.**

Try to refactor your previous solution to a pattern. Then repeat with another. Or start from scratch considering patterns.

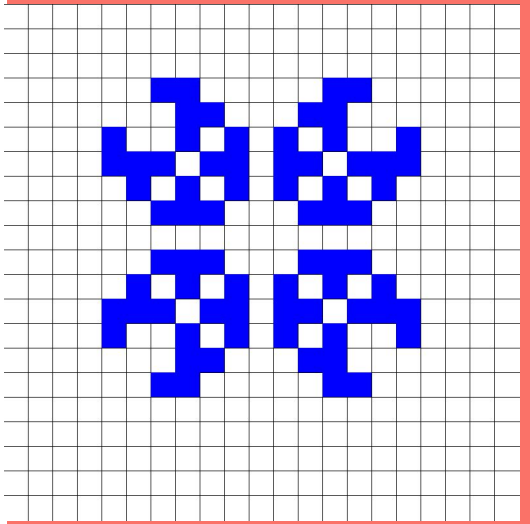
Challenge: Add the following features

- **Provide an “undo” command, identified by “U”**
- **Add logging when a command is executed**



The Game of Life

The universe of the Game of Life is an infinite, two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead, (or populated and unpopulated, respectively). Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:



- Any live cell with fewer than two live neighbours dies, as if by underpopulation.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

Give it a try, and then refactor to patterns!