Summary and the list of assumptions

I've tried to separate and isolate different logic layers to their respective classes so here’s what I’ve come up with.

I use one class named RobotController to control everything regarding the robots in the solution. It allows to input commands which will setup zone, create and setup robots and then run the instructions with a single Run() call. While it is possible to implement real-time service solution to process input commands on the fly, I assume that such complex solution will take more time to develop and is a subject for an improvement.

RobotController keeps tracking of all the robots and danger zones. It is also responsible for robot command interpretation and command result handling. This approach leaves all complex logic to the controller and allows us to implement wide variety of robots easily. Robots are created using factory so controller do not know about their exact implementation. CommandState enum has been chosen instead of simple boolean flip flag to control which command we’re expecting to receive now. It allows us to add more steps into command input handling process if needed.

Default robot class consists of 3 layers. First one is IRobot<T> generic interface. It allows us to supply only viable data for the end clients hiding the exact implementation. Generic T type allows us to specify command type for the robot. By default, the command is represented by the single “char” symbol and commands are stored inside the robot using Queue.

Robot itself is not dependent on exact command type which is a good call for future extensibility. It simply stores its own command queue. Fetched commands are processed by RobotController and then are issued back to the robot through available interface methods (or advanced commands, explained later).

The second robot layer is a RobotBase<T> abstract class which represent a solid base for any kind of robot we can possibly create. It contains all the core logic which is used by all robots. The final 3rd layer is represented by an actual Robot<T> class based on the RobotBase<T>. It contains more specific robot logic which is implemented using the override functionality. For example, we can override move and rotation methods if we would like to implement diagonal movement. This is why I use degrees for rotation internally and provide methods to interpret them as NWSE symbols as task require.

I also want to mention a piece of logic I’ve implemented to help improve commands handling. By default, core commands like rotation and movement are bound to interface methods. If we want to add new commands it pose some problems because we have to update interface and its implementation in all robot classes and corresponding calls in controller. I’ve left initial commands code intact but added some new functionality with examples on how we can mitigate this issue. Take a look at AdvancedBot class implementation. Here we use a dictionary to store command implementations inside the robot. Each robot decides which command collection it want to use. We can use overrides to core robot params like Rotation and Position to make them return data from advanced commands instead or even use a new method ExecuteAction on robot to try execute the command if robot support it. The system is quite simple in its current state and do not provide good performance but there’s always a way for the improvements.

Things which falls out of the scope and I would improve

1. Robot identification for better logging and maintainability
2. Better logging code instead of a simple console logging
3. Improve code and make it resilient to multithreading operations. It will run as a background service and allow to input commands in the real-time. This approach is a time consuming in terms of development but brings a lot of flexibility to solution.
4. We can also improve RobotController to support custom factory (easy) to allow end users to create and use custom bots if needed.

Time estimation

A time estimation is quite tricky. For the basic functionality I will say 6 hours plus 2 hours for testing and polishing. If we talk about improving this code to support multithreading it will probably take another 4 hours of intensive work and testing.