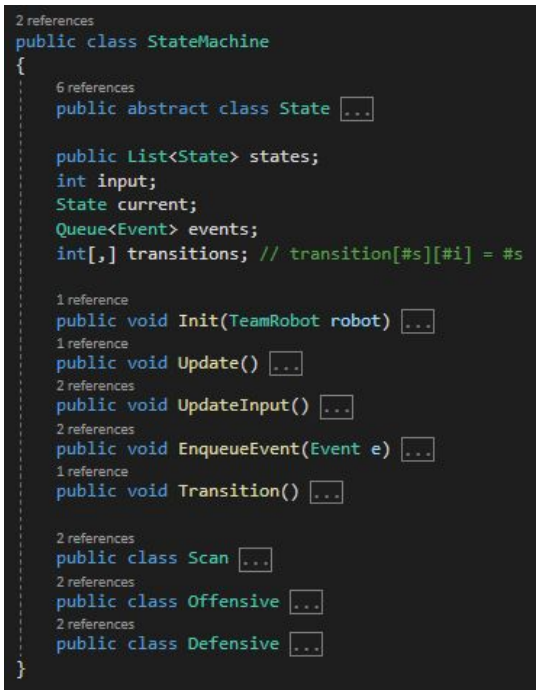
WreckItRalph Design Document

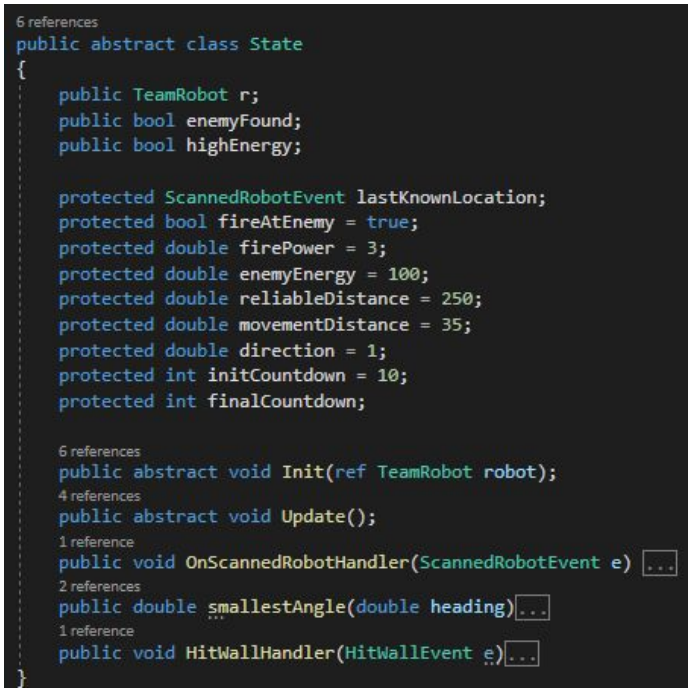
By Jordan Smith for CAP4053, Spring 2020

Finite State Machine Architecture

As per the specifications set for this project, I was required to use a robust, extensible, abstracted decision-making architecture for the robot's AI, so I chose to use a finite state machine (FSM) to manage my robot's behaviors. My FSM's structure is more or less the same as the professor's example FSM that he provided in the lecture:



```
2 references
public class StateMachine
{
    6 references
    public abstract class State ...

    public List<State> states;
    int input;
    State current;
    Queue<Event> events;
    int[,] transitions; // transition[#s][#i] = #s

    1 reference
    public void Init(TeamRobot robot) ...
    1 reference
    public void Update() ...
    2 references
    public void UpdateInput() ...
    2 references
    public void EnqueueEvent(Event e) ...
    1 reference
    public void Transition() ...

    2 references
    public class Scan ...
    2 references
    public class Offensive ...
    2 references
    public class Defensive ...
}
```



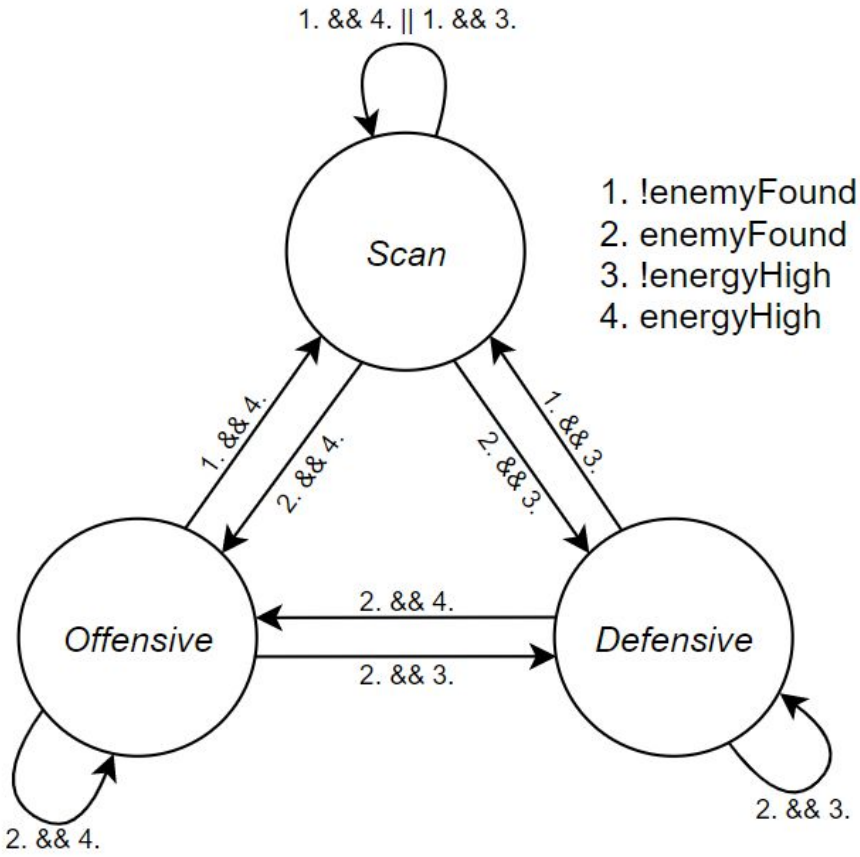
```
6 references
public abstract class State
{
    public TeamRobot r;
    public bool enemyFound;
    public bool highEnergy;

    protected ScannedRobotEvent lastKnownLocation;
    protected bool fireAtEnemy = true;
    protected double firePower = 3;
    protected double enemyEnergy = 100;
    protected double reliableDistance = 250;
    protected double movementDistance = 35;
    protected double direction = 1;
    protected int initCountdown = 10;
    protected int finalCountdown;

    6 references
    public abstract void Init(ref TeamRobot robot);
    4 references
    public abstract void Update();
    1 reference
    public void OnScannedRobotHandler(ScannedRobotEvent e) ...
    2 references
    public double smallestAngle(double heading)...
    1 reference
    public void HitWallHandler(HitWallEvent e)...
}
```

The *StateMachine* class has *States* that it transitions between based on the current *State* (*current)* and input (*input*). In order to update the input within the Robocode framework, event handlers are used in conjunction with a queueing of events in order to parse all incoming information about the state of the current battle, such as if the robot has bumped into a wall or spotted an enemy robot recently. Every turn of the battle, the current state and input are updated, and based on the updated state and input a transition to a state is determined.

The *State* class is where the individual behaviors of the robot are determined based on the values of the two booleans *enemyFound* and *highEnergy*, which dictate whether an enemy is being tracked and if our health is at or above 50%, respectively. Each *State* is initialized with *Init()*, which stores the reference to the robot for controlling it and sets *State*-dependent variables. The *Update()* function is where some behavior happens, but most behavior is handled by the *OnScannedRobotHandler()* method, where the main strategy behavior of WreckItRalph is

written. These *Init()* and *Update()* methods are abstract to allow the child states (*Scan, Offensive, Defensive*) to initialize and update themselves according to their unique behavior, whereas the *OnScannedRobotHandler()* is inherited by all child states as the targeting and movement is identical between all states. The other two methods, *smallestAngle()* and *HitWallHandler()* are effectively helper functions for *OnScannedRobotHandler()*.

States & Behaviors



WreckItRalph has three states: *Scan*, *Offensive*, and *Defensive*. The transition states based on *enemyFound* and *energyHigh* are displayed in the diagram above. To sum it up, the FSM initiates with the *Scan* state, which spins the radar to search for an enemy. When an enemy is found, a transition will occur to the *Offensive* state if the robot's health is above 50% or to the *Defensive* state if the robot's health is below 50%. The *Offensive* state simply circles the enemy with a variety of stop-and-go movement [1] where the robot will move a certain distance when a bullet is fired, in order to mess up the enemy robot's targeting, predictive or otherwise. The robot also shoots at the enemy robot using linear targeting [2]. The *Defensive* state is identical to the *Offensive* state, except that it will not fire at the enemy beyond a certain distance, and its fire

power is based on the enemy's distance to the robot, increasing as the enemy draws near and decreasing as it moves away from the robot.

Strategy

Overview

WreckItRalph's general strategy is one of circling and strafing. The robot will circle an enemy, moving only in small strides to evade incoming bullets whilst maintaining the highest chance of hitting the enemy if it is assumed that the enemy is traveling towards the robot. Through trial and error, I arrived at the conclusion that the shortest distance possible to maintain this behavior is around 35 px, which is just enough in most situations to barely dodge the incoming bullets. When circling, the robot may bump into a wall, so if it does it reverses the direction it is circling between clockwise and counterclockwise around the enemy.

Specifics

Targeting

WreckItRalph targets the enemy with its radar by computing the difference between its body's heading and its radar heading and adding the enemy's bearing in order to determine the direction the radar should be aiming at to stay locked on to the enemy.

Firing

WreckItRalph's firing angle is the same as the radar's targeting angle when the gun cannot fire, either due to low energy, gun heat, or if the enemy is too far away when in the *Defensive* state. If the robot is able to fire, then a prediction is made based on the assumption that the enemy robot will keep moving in the same direction at the same speed in order to fire at where it will be in the future. It is not perfect though, as it is easily fooled by any type of movement that isn't essentially moving in straight lines.

Movement

WreckItRalph's movement is to circle the enemy in small distances, only walking when the enemy has fired a bullet. It utilizes the Stop and Go movement strategy by monitoring the enemy's energy at all times, and only calling setAhead() when the enemy's energy drops. This is not a perfect option, as my implementation does not take into account any drop in energy as a result of my robot's attacks, but it works well enough. The circular movement is accomplished easily by taking the body heading and adding 90 degrees to it to make the robot always move perpendicular to the enemy robot, which effectively makes it circle the robot. I chose to use a circular movement instead of head-on or oscillator movement because I found that it worked the best in my trials with all three tactics. I believe this to be because the circular strategy maintains

distance between the robots, as WreckItRalph never directly approaches the enemy robot, allowing for more room for dodging which synergizes well with the stop-and-go movement tactic.

References

[1] Stop And Go: https://robowiki.net/wiki/Stop_And_Go
[2] Linear Targeting: https://robowiki.net/wiki/Linear_Targeting