SOLOMON

AN ARTIFICIAL INTELLIGENCE SYSTEM BY

CARL LANGE

AND

CIARÁN MCCANN

2010

Carl Lange & Ciarán McCann
C00138612 & C00134748
February 17, 2010

SYSTEM DESIGN DOCUMENT - SOLOMON (ROBOCODE)

Introduction

Solomon is an Artificial Intelligence robot, whose main feature is the ability to choose a tactic based on its situation. Solomon has a library of approximately eight tactics, (the number of which can be easily added to), and will pick the best tactic for its current situation. Solomon was coded and designed in fewer than ten days in January 2010.

It's based off of the ideas of a finite state machine:

> "A **finite state machine** (**FSM**) or **finite state automaton** (plural: *automata*), or simply a **state machine**, is a model of behaviour composed of a finite number of states, transitions between those states, and actions. It is similar to a "flow graph" where we can inspect the way in which the logic runs when certain conditions are met. A finite state machine is an abstract model of a machine with a primitive (sometimes read-only) internal memory."
>
> ~ Wikipedia (http://en.wikipedia.org/wiki/Finite-state_machine)

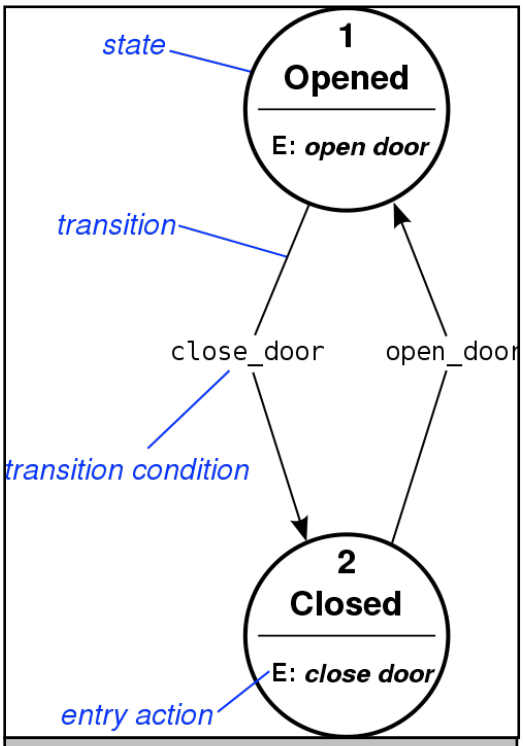


Diagram of a simple finite state machine.

The AI assesses its health and chooses which state it's in, i.e extremely aggressive, aggressive, defensive, or extremely defensive. These states have certain behaviours (henceforth called tactics) associated with them. These tactics are used, assessed for efficiency, and used again based on their past efficiency.

Essentially, this means that we can have as many different behaviours as we like, and Solomon will pick the best one for the situation.

AI

Solomon's AI class is, simply put, the brain of the robot. Though it doesn't contain much code, it performs vital functions in decision making. Solomon is based on a finite state machine, in that it will perform certain actions based on the state of a variable (such as health, in our case). Originally we had wanted to base it on multiple variables such as health, enemy health, score etc, unfortunately our 10 day timeframe did not permit this to be implemented.

We have 4 statuses: extremely aggressive, aggressive, defensive and extremely defensive. Inside each status there are about between two and four tactics which reflect the status. For example, the extremely aggressive tactics tend to fire with much more power, move close to enemy robots, and generally don't display any regard for the robot's own wellbeing. Conversely, extremely defensive tactics rarely fire at all and keep their distance as much as possible.

Solomon has a memory of each tactic and its efficiency in the passed. Our original idea was to write this info to a text file so we could recall it between matches with the same opponent. However, it seems we can't do this without using the AdvancedRobot class, the use of which is not permitted in this competition. (Side note: The rules seem to contradict themselves in this instance.)

The tactics are rated by the change in health of Solomon, during a very short set period of time (at the time of writing, this is 0.9 seconds). If the health has changed negatively during that time, a zero is recorded, and if it has changed positively (that is, if the health has risen or dropped only very little), a one is recorded.

From this sequence of ones and zeroes, isGoodTactic in the CTactic class, below, will determine whether or not the tactic is good or bad and whether to switch out of it to another tactic.

CTactic Class

The CTactic class, which is the superclass for every tactic, contains some methods, which are described in gruelling detail here.

run_(solomon s)

This is an empty method, put here for overriding. Takes in Solomon so that it can, if necessary, call methods only available to Solomon. This can be said about almost all methods hence.

onScannedRobot_(solomon s, ScannedRobotEvent e)

onHitByBullet_(solomon s, ScannedRobotEvent e)

Same as run_(solomon s).

fire(solomon s, double enemyDist)

This is a dynamic firing method, and personally, my second-favourite feature of the entire robot. It modulates the power of the bullet to be fired (between 0.1 and 3.0, the global limits). The power is calculated by two things: the enemy distance and the bias, which is decided (through a case statement) by the status of the robot (aggressive, defensive, et cetera). If Solomon's in extremely aggressive mode, it'll be much more likely to fire with full strength, and if it's in extremely defensive mode, it will almost never fire with full strength.

isGoodTactic(int status)

//FIXME: This is called by pickTactic in the AI class. It adds up every item in the gaugingList (which is an ArrayList of bytes, just ones and zeroes, based on the difference in health in the last 0.9 seconds), divides the total by the size of the list. This gives it the average of zeroes and ones (between 0.0 and 1.0). Then, it checks whether

this result is greater than the gauging threshold. If it is, it returns true (i.e. That the tactic is good). If not, the tactic is not good, and returns false.

getRandom()

Simply returns a random number. Done here to combat code duplication.

getRandom(int n)

Returns a random number between zero and n.

[There are also a few simple radian translations of calculations that would otherwise return degrees. This is done for compatibility with almost every method in the native Math class.]

Having the superclass laid out this way, with many protected methods (and protected objects, such as a Random object that can be used rather than unnecessarily using up memory creating more of them), makes it extremely easy to add tactics to the tactic library, also preventing code duplication and unnecessary complications to do with naming, et cetera.

Tactics Library

Because of the modularity in tactics, it's very easy to add or remove tactics from the robot. This means that tactics can easily be tested and migrated into the library.

The tactic library is a two-dimensional array. It's filled (manually) with the tactics, during Solomon's construction (that is, when a match starts). Each tactic in the library overrides several methods in the CTactic class, such as run_, onScannedRobot_ et cetera. These methods have underscores behind their names as they otherwise would share a name with methods in Solomon's core, and we'd like to avoid confusion.

We tried to keep the tactics fairly simple, as we only had ten days to design and code the entire robot, three of which we spent designing, and five of which were part of Carlow I.T.'s Raise And Give week. We learned much from the sample code included with robocode, specifically to do with targeting and some fundamental mathematics. Neither Ciarán nor I are particularly excellent at maths, but both of us are (somewhat) competent programmers (we hope you agree), and thus, the AI, rather regrettably, has considerably more polish than the tactics themselves. Nevertheless, the ability to switch between tactics is an advantage that we feel others may not have.

Software and Tools

Both Ciarán and I used the Eclipse IDE to develop Solomon, as the editor built into Robocode seems to be little more than a very basic text editor with syntax highlighting. As we both use a multitude of Operating Systems (Mac OS X, Ubuntu Linux, and Windows, if necessary), it was helpful to have a unified UI. We used a Subversion repository (the log of which can be viewed at http://code.google.com/p/robocodecarlow2010/updates/list) on Google Code to deal with version control and to prevent issues with merging code et cetera.

Much software designing was done with pens and paper, and some of these pages can be found attached to this document, not that they're legible.

CTactic Table	0	1	2
State 1 Extremely Aggressive	Moves towards the enemy and fires a lot (every 50 units that it moves). A short tutorial used to learn about non-iterative linear targeting. http://bit.ly/bw2s2r	//TODO	//TODO
State 2 Aggressive	//TODO	//TODO: Spins and fires. Very simple tactic.	Acts very like sample.fire, in that it rotates a small amount and stops on the enemy. However, it implements bullet strength modulation, making it much better.
State 3 Defensive	Acts very like wallkiller, that is, Solomon follows the walls and fires inwards. Scans in the corners. Same targeting as ea0.	//TODO	//TODO
State 4 Extremely Defensive	TODO: Longest-distance random movement. That is, it consistently moves to a random spot where it is further from the main opponent (within reason).	//TODO	//TODO

CLASS DOCUMENTATION

itc.solomon Class Reference

Public Member Functions

byte getStatus ()
void setStatus(byte status)
solomon ()
void run()
void onScannedRobot(ScannedRobotEvent e)
void onHitByBuller(HitByBulletEvent e)
void onHitRobot(HitRobotEvent e)

Detailed Description

Solomon - a robot by IT Carlow students Ciarán McCann and Carl Lange.

Constructor & Destructor Documentation

itc.solomon.solomon ()
The constructor for Solomon. Essentially only calls populateLibrary.

Member Function Documentation

byte itc.solomon.getStatus ()
Simply returns the current status (0,1,2,3).
Returns:
currentStatus.

void itc.solomon.onHitByBullet (HitByBulletEvent e)
What to do when you're hit by a bullet. This calls the current tactic's onHitByBullet_() method.

void itc.solomon.onHitRobot (HitRobotEvent e)
What to do when you hit another robot. This calls the current tactic's onHitRobot_() method.

void itc.solomon.onScannedRobot (ScannedRobotEvent e)
What to do when you see another robot. This calls the current tactic's onScannedRobot_() method.

void itc.solomon.run ()
This sets Solomon's colours, then goes into an infinite loop. This loop is Solomon's main loop (its "game loop", of sorts.

void itc.solomon.setStatus (byte *status*)
Simply sets the current status (0,1,2,3).

itc.AI Class Reference

Static Public Member Functions

static byte getGaugingThreshold ()
static void setGaugingThreshold (byte gaugingThreshold)
static int pickTactic (int status, int currentTacticIndex, CTactic tacticLibrary[][])
static byte gaugeTactic(double healthBefore, double currentHealth)

Static Public Attributes

static byte gaugingThreshold = 3

Detailed Description

This class is Solomon's decision making component. Here it assesses the past experiences and judges whether the tactic was successful in the past.

Member Function Documentation

static byte itc.AI.gaugeTactic (double *healthBefore*, double *currentHealth*) **[static]**

Returns a one or a zero based on the health change in the time before a tactic was used and after to determine if it's a successful tactic

static int itc.AI.pickTactic (int *status*, int *currentTacticIndex*, CTactic *tacticLibrary*[][]) **[static]**

Picks a tactic for the robot based on the current status it's in i.e defensive, aggressive etc. It then calls another method isGoodTactic() which will find if the tactic as been successful in the past

Returns: Tactic to be used

Member Data Documentation

byte itc.AI.gaugingThreshold = 3 **[static]**

This is the percentage amount of negative change which is required or greater to get a negative result on the efficiency.

itc.CTactic Class Reference

Public Member Functions

void run_ (solomon s)
void onScannedRobot_ (solomon s, ScannedRobotEvent e)
void onHitByBullet_ (solomon s, HitByBulletEvent e)
void onHitRobot_ (solomon s, HitRobotEvent e)
boolean isGoodTactic (int status)

Protected Member Functions

void fire (solomon s, double enemyDist)
double getRandom ()
double getRandom (int highest)
void turnGunRightRadians (solomon s, double amountToRotateRadians)
void turnRadarRightRadians (solomon s, double amountToRotateRadians)
void turnRightRadians (solomon s, double amountToRotateRadians)
double getHeadingRadians (solomon s)
double getGunHeadingRadians (solomon s)
double getRadarHeadingRadians (solomon s)
double convertToRadians (double degrees)
double convertToDegrees (double radians)

Protected Attributes

final double GAUGING_THRESHOLD = 0.7
List< Byte > **gaugingList** = new ArrayList<Byte>()
Random **r** = new Random()

Member Function Documentation

double itc.CTactic.convertToDegrees (double *radians*) [protected]

Takes in radians and converts to degrees.

double itc.CTactic.convertToRadians (double *degrees*) [protected]

Takes in amount in degrees and returns it in radians

void itc.CTactic.fire (solomon *s*, double *enemyDist*) [protected]

Uses the distance of the enemy robot to figure out how much energy to expend when firing. The bias varies, depending on the robot's status. If it's very aggressive, it'll be more likely to fire with full strength. If very defensive, it'll almost never do that, and so on, so forth.

double itc.CTactic.getGunHeadingRadians (solomon *s*) [protected]

This should be overwritten by every tactic.

double itc.CTactic.getHeadingRadians (solomon s) **[protected]**
Gets the heading in radians.

double itc.CTactic.getRadarHeadingRadians (solomon s) **[protected]**
Gets the radar heading in radians.

double itc.CTactic.getRandom (int *highest*) **[protected]**
Returns a number, between zero and input.

double itc.CTactic.getRandom () **[protected]**
Returns a random number.

boolean itc.CTactic.isGoodTactic (int *status*)
Calculates the efficiency of the tactic using the gaugingList and returns true or false

void itc.CTactic.onHitByBullet_ (solomon s, HitByBulletEvent e)
This should be overwritten by every tactic.

void itc.CTactic.onHitRobot_ (solomon s, HitRobotEvent e)
This should be overwritten by every tactic.

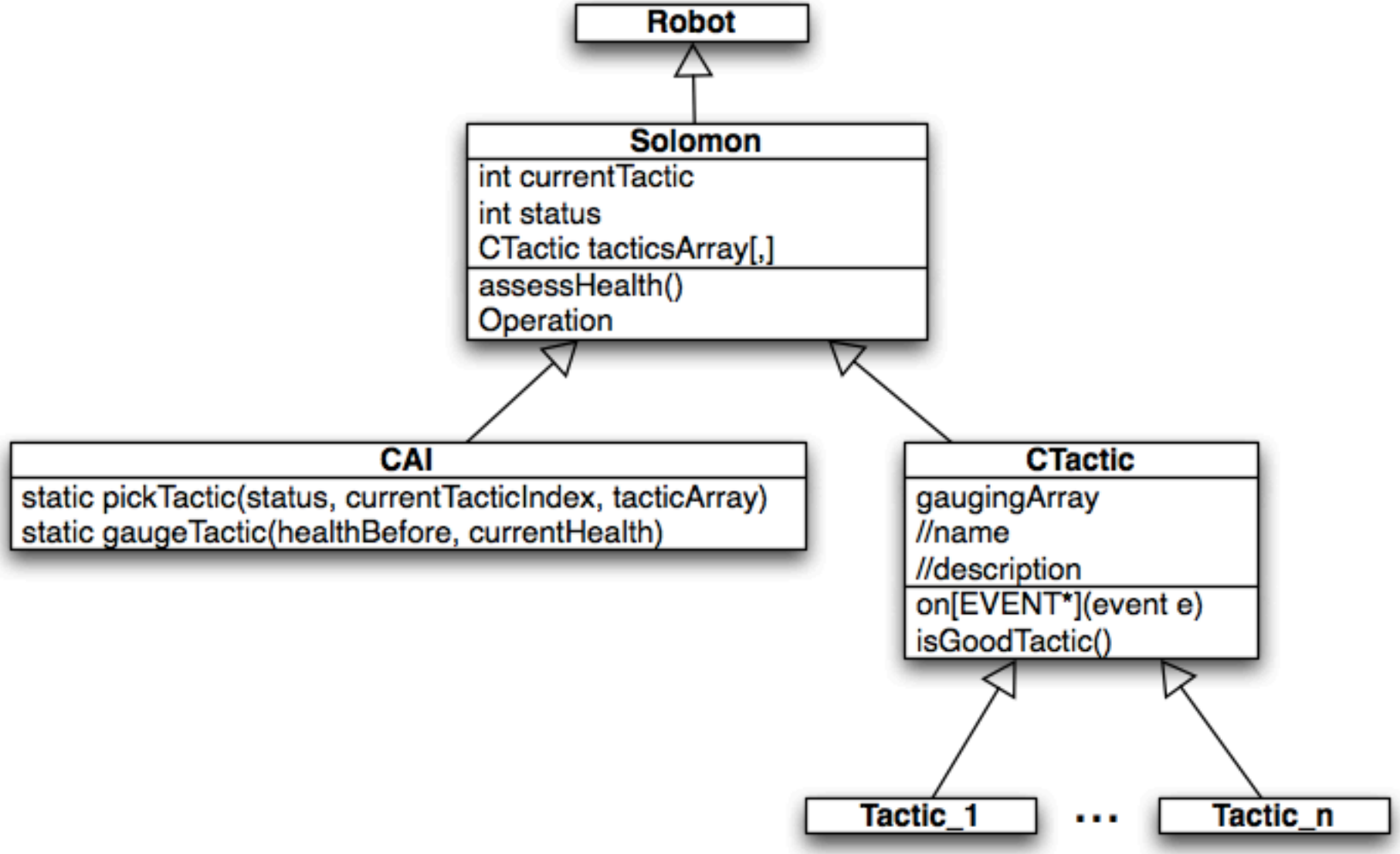
void itc.CTactic.onScannedRobot_ (solomon s, ScannedRobotEvent e)
This should be overwritten by every tactic.

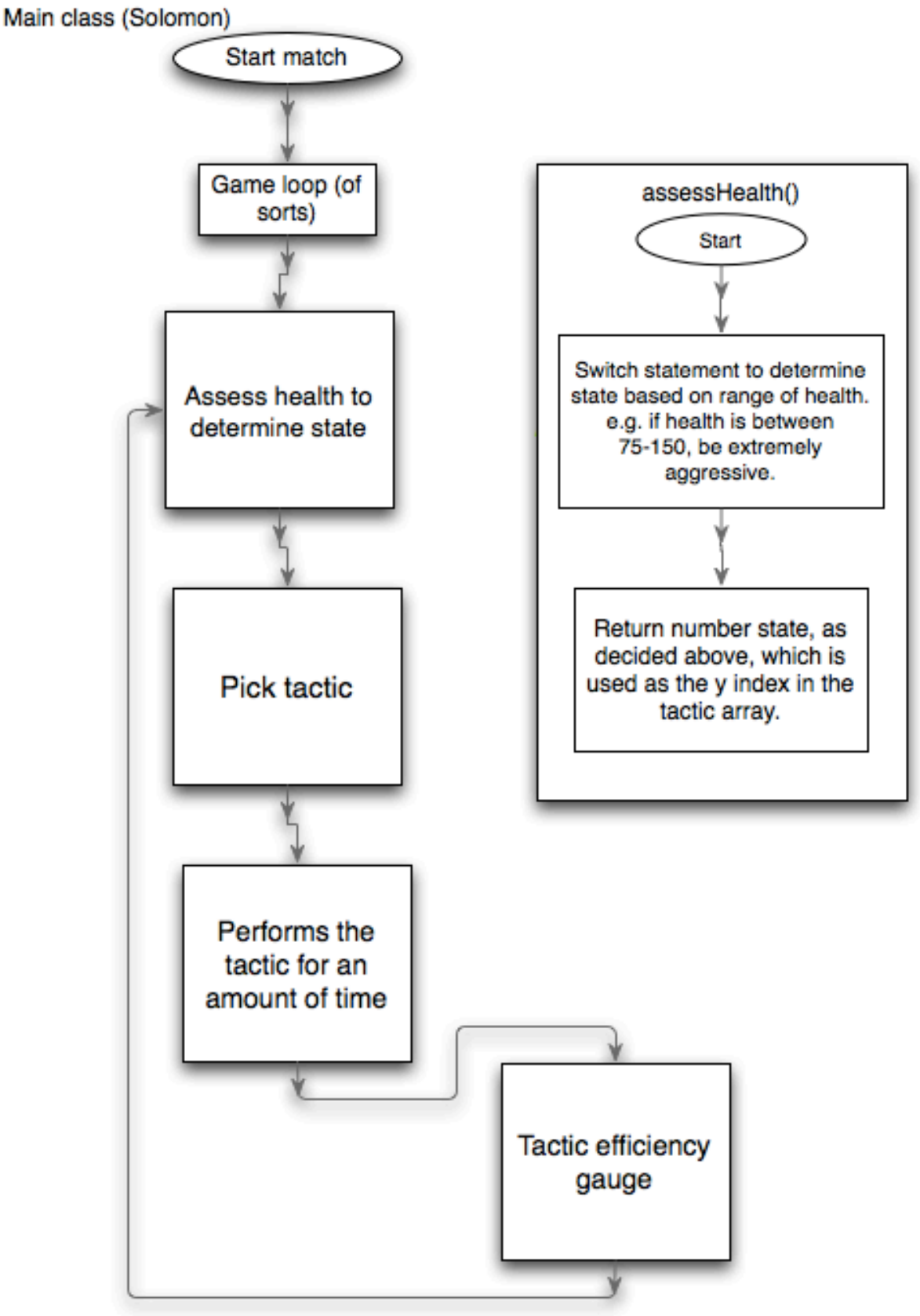
void itc.CTactic.run_ (solomon s)
This should be overwritten by every tactic.

void itc.CTactic.turnGunRightRadians (solomon s, double *amountToRotateRadians*) **[protected]**
Turns the gun right by the specified number of radians.

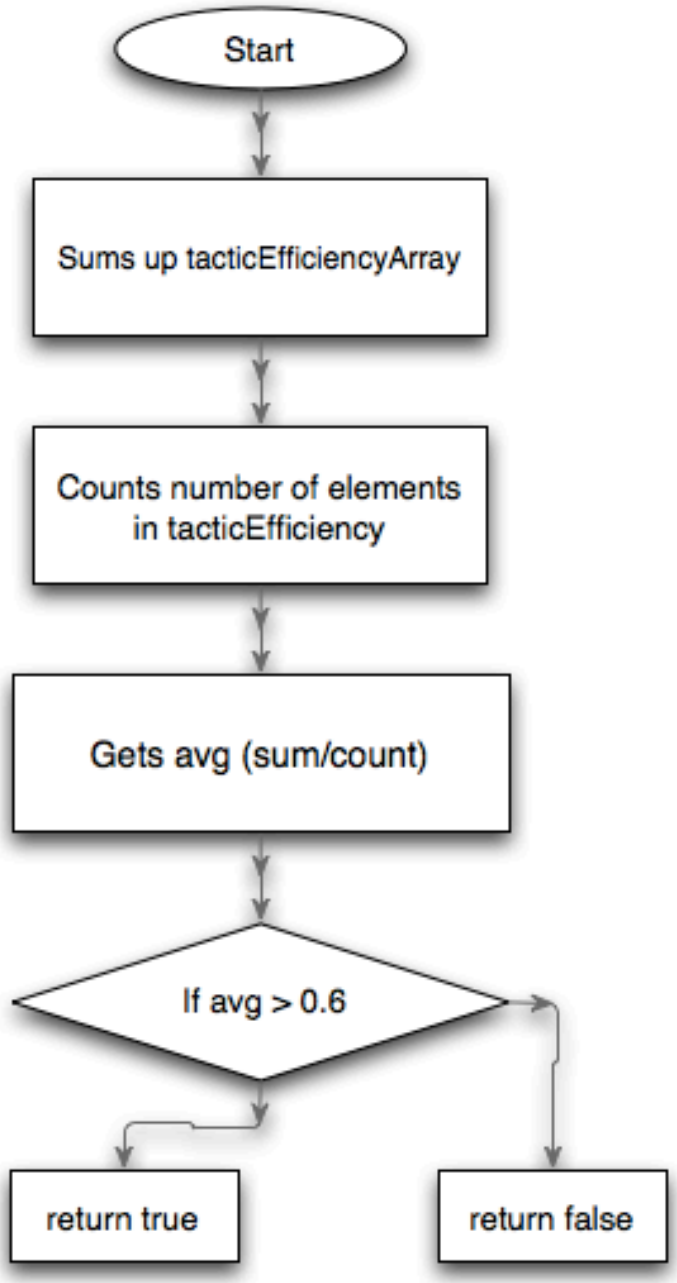
void itc.CTactic.turnRadarRightRadians (solomon s, double *amountToRotateRadians*) **[protected]**
Turns the gun right by the specified number of radians.

void itc.CTactic.turnRightRadians (solomon s, double *amountToRotateRadians*) **[protected]**
Turns the whole robot right by the specified number of radians.

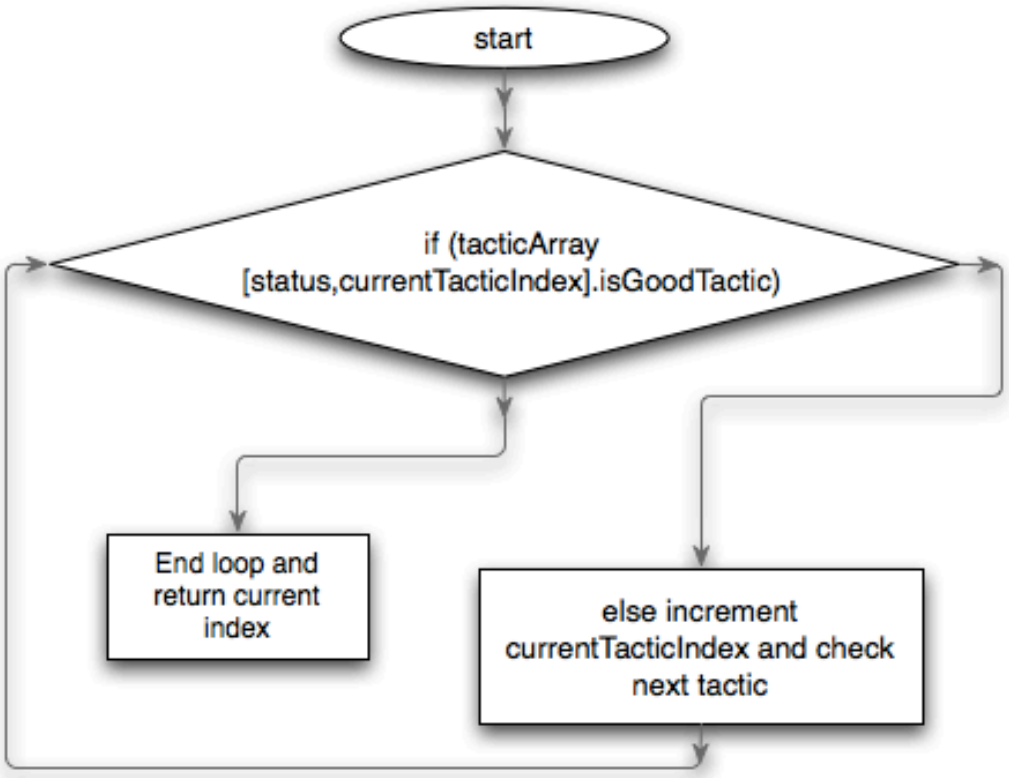



Main class (Solomon)
Start match
Game loop (of sorts)
Assess health to determine state
Pick tactic
Performs the tactic for an amount of time
Tactic efficiency gauge
assessHealth()
Start
Switch statement to determine state based on range of health. e.g. if health is between 75-150, be extremely aggressive.
Return number state, as decided above, which is used as the y index in the tactic array.

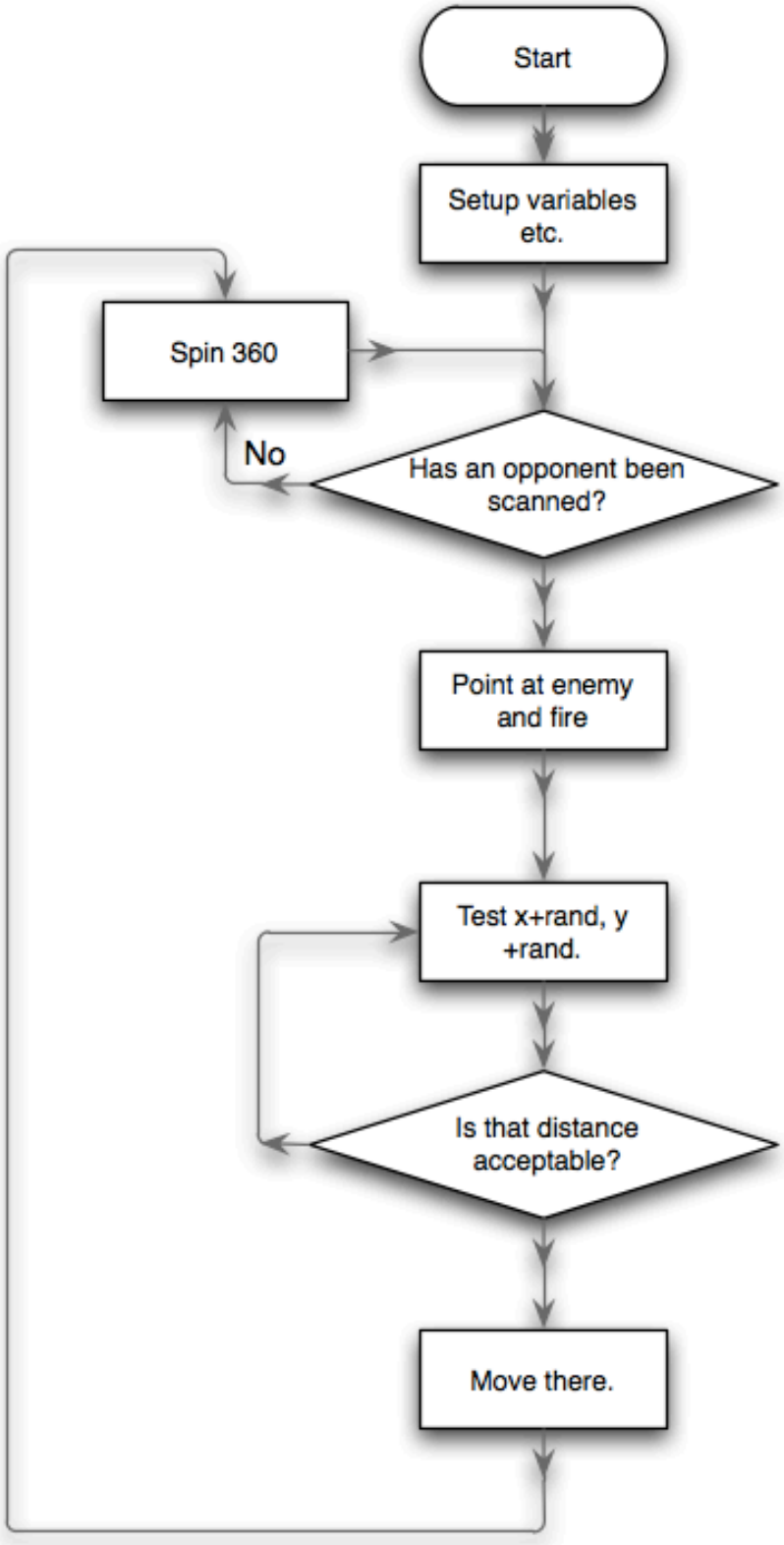
isGoodTactic()
in Tactic class



pickTactic(status, currentTacticIndex, tacticArray)



ed0


Start
Setup variables etc.
Spin 360
No
Has an opponent been scanned?
Point at enemy and fire
Test x+rand, y +rand.
Is that distance acceptable?
Move there.