nex+Gen Griffin Robot Software Architecture

Griffin Robotics, Team 7582

Introduction

Over the past few competitions, the Griffin Robotics team has been faced with several challenging problems. According to the rules of the FTC competition, of course, the team must provide a Driver Op Mode for the 2-minute driver controlled portion of the competition, and one or more Autonomous Driver Op Modes for the 30-second autonomous portion. The autonomous portion starts a round, followed by the driver portion. These requirements are common to all teams in the competition.

There are, however, additional goals that were important to our team. These goals are not necessarily unique to our team, other teams may share these goals as well. But, they make the task of designing and implementing the Op Modes more challenging. To satisfy these goals we've created several useful innovations. Some of those innovations take advantage of subtle features of the FTC code base in unique and novel ways. Other innovations solve problems using techniques that are standard in the industry, but because they are more advanced techniques, we include them here as well.

In this document we describe in detail the software architecture that implements these goals. In this document we describe the architecture itself without fully justifying why it is organized this way. A companion document, *nex+Gen Griffin Robot Software Innovations*, calls out the individual innovations, how they are implemented, and ties those innovations to the architecture described here.

General Approach

This design uses the general software technique of *divide-and-conquer* to separate different robot functions into individual discrete components. In order to do this we make extensive use of *encapsulation*, *abstraction* and *inheritance*, three of the four main features of all object-oriented languages, like Java. (The fourth feature is *polymorphism*, which is also a powerful concept, but it is not needed for this robot.)

Abstraction is used to create individual components, such as our robot base and robot arm assemblies. Each component is separated into two classes, one that describes the interface, and one that describes the implementation. Conceptually, the interface describes *what* the component does, while the implementation describes *how* the component does it. Inheritance is used to link the two parts together. Abstract functions in the interface are used extensively to allow the different Op Modes to use the robot components without knowing what the implementation really is. A third component of the robot, of course, is the Op Mode that drives the robot.

Implementing the robot in this manner has several advantages. First, it divides the robot into logical entities that are independent of each other. The arm, for example, can easily be thought of as a separate component from the base. The Op Mode is separate from both the base and the arm. Second, it allows components to be mixed and matched as needed, which makes it possible to support multiple robots without duplicating common pieces of code.

Infrastructure

Our design divides the robot infrastructure into two main components, the robot base, and the hook and arm assembly. The base is responsible for moving the robot around the field, while the hook and arm assembly grabs onto stones and foundations. As mentioned in the previous section, each component is divided into two sections, an interface section and an implementation section. The interface section declares what functions can be used by an Op Mode to operate that component. The implementation section defines how the robot uses the various motors, servos, etc., to implement each function. The interface section is implemented as an abstract parent class, while the implementation section is implemented as a concrete (non-abstract) child class of the parent.

Robot Base

We use a holonomic design for our robot base. A holonomic drive allows the robot to move in any direction without first changing its orientation. A common example of this is a forklift used in a warehouse. Warehouses often have narrow aisles so the forklift has to be able to maneuver in all directions as flexibly as possible. A holonomic drive makes that possible. The high-level structure for our software implementation of this component is shown in Figure 1.

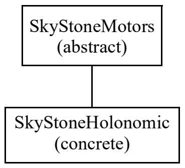

Figure 1 - Robot Base High-Level Structure

The `SkyStoneMotors` class consists of 35 lines of code, including declarations for five abstract functions. The abstract functions are:

```
public abstract void init();
public abstract void move(double bearing, double power, double turn);
public abstract void move_to(double bearing, double power, double inches);
public abstract void turn_to(double bearing, double power);
public abstract void stop();
```

The functions `init` and `stop` initialize and shut down the robot devices, respectively. The `move` function is used during the Driver Op Mode to propel the robot in whatever direction and speed the driver wishes. The `move_to` and `turn_to` functions are used during Autonomous Op Modes to propel the robot specific distances and directions. For the move and turn functions, `bearing` is the relative bearing of travel, in degrees, where 0° is forward, 90° is to starboard (to the right), 180° is aft (toward the rear), 270° or -90° is to port (to the left), and so on. The `power` value is a real number between -1 and +1, and reflects the relative power going to the drive motors, scaled to reflect the direction of travel. The `turn` value is similar, but it is applied to all wheels uniformly to cause the robot to turn. The value `inches` is the distance, in inches, the robot will travel. Each of these functions must be implemented by the child class (`SkyStoneHolonomic`).

The parent class also includes a few additional items, such as a pointer to an object of type `OpMode` (`op_mode`), a constructor, and a sleep function. The object pointer is needed because it provides a necessary execution context into the FTC software that allows the robot to make a connection to the motors, servos, and other hardware components. The constructor in the parent class is used by the child class in its constructor to assign the object pointer.

The `SkyStoneHolonomic` class provides a concrete implementation for each of the functions defined in the `SkyStoneMotors` parent class. It translates the desired behavior of each function, such as "move to a specific location" for the `move_to` function, into specific commands to each of the motors in the base.

As mentioned before, the base uses a holonomic design. That means the `move` and `move_to` commands must translate a bearing value into direct commands to the motors. We use dead reckoning to navigate to the new location, that is, we turn each of the wheels a calculated number of times with the expectation that once completed the robot will be in the new position. We base our calculation on a combination of measurements taken on the field and simple trigonometry. This works well enough in practice, but it is based on a number of assumptions, including:

1. The amount of traction between the wheels and the rubber surface of the field
2. Distribution of weight in the robot
3. Uniformity of power going to each of the wheel motors, etc.

If any of our assumptions is incorrect or inaccurate, for example, the traction changes between the practice field and the competition field, or the addition of ballast changes the distribution of weight in the robot, dead reckoning will move the robot to the wrong place on the field. The use of sensors and feedback could correct this, but we don't use them at this time.

Implementing the `move` and `move_to` functions requires first, a translation from the normal robot coordinate system, where 0° is forward and 90° is in a clockwise direction, to the mathematical coordinate system, where 0° is to the right and 90° is anti-clockwise. In our case the Java *sine* and *cosine* functions are in radians, so we must also convert from degrees to radians. For some bearing β in degrees to some angle α in radians, this translation is:

$$\alpha = \pi/2 - \pi\,\beta / 180$$

Each of the four wheels needs a different equation to determine the power going to the wheel, based on its position in the base and the intended direction of travel. If the turn component is non-zero, it is added directly to the power value for each wheel. There is more than one possible choice for these functions, but the functions that give the behavior we expect, are:

$$\begin{aligned}
power_{front\ left}(\beta) &= -sin(\pi/4 + \pi\,\beta / 180) + turn \\
power_{front\ right}(\beta) &= cos(\pi/4 + \pi\,\beta / 180) + turn \\
power_{back\ right}(\beta) &= sin(\pi/4 + \pi\,\beta / 180) + turn \\
power_{back\ left}(\beta) &= -cos(\pi/4 + \pi\,\beta / 180) + turn
\end{aligned}$$

Power values are limited to between ±1. If one exceeds that, they must all be scaled uniformly.

The `move` function uses the motors a little differently than `move_to` and `turn_to`. The `move` function does not use the motors to go to a specific encoder setting. It operates the motors using the `RUN_USING_ENCODER` mode. It turns the motors on and leaves them on until another user command changes the power to some other setting.

In contrast, `move_to` and `turn_to` advance each motor to a specific encoder setting using the `RUN_TO_POSITION` mode. They must reset the encoders to zero, set encoder targets for each motor, then apply power to all of them at the same time. Furthermore, the functions must wait until each motor has reached its target before it returns control back to the calling routine. The motor power settings are determined using the same equations as are used in the move function, and the encoder targets are computed very similarly.

The final consideration is that the target distances are given in units of inches, but the motors are moved in units of encoder clicks, so a conversion factor has to be provided. The conversion factor depends on the circumference of the wheels and the slippage of the wheels against the field. Each rotation of a wheel represents four encoder clicks multiplied by any gear reduction that takes place. Our wheels use a 40:1 gearing, so one rotation represents 160 encoder clicks. Our wheels have a diameter of 4" which gives a circumference of a little over 12.5". In other words, 160 encoder clicks represents roughly 12.5", or 12.8 encoder clicks per inch.

The wheels are also mounted at a 45° angle, with rollers to allow easy slippage during travel at an angle, on a mat that has a surface with imperfect traction. These factors are difficult to incorporate individually, so the final conversion factor needs to be determined experimentally. For our OwO robot, that has Omni wheels, the ratio is 100 encoder clicks per inch. Our UwU robot, with its Mecanum wheels, seems to do best with a ratio of 131 encoder clicks per inch.

A similar determination must also be used to convert degrees of rotation to encoder clicks, used when the robot turns to a new heading.

These ratios are provided and stored with the object when the Op Mode first creates an object that represents the motorized base. These ratios, along with the `OpMode` context, are provided to the `SkyStoneHolonomic` constructor. There is also a fourth value, namely the drift in degrees, which is used to compensate for the natural drift of the robot to one side or the other. If the robot naturally drifts 5° to port, then set the drift value to 5° to starboard.

Hook and Arm Assembly

The hook and arm assembly is really three separate components, the hook, arm and claw, grouped together for convenience. Like the robot base, the software is organized as a parent class that declares the interface (`SkyStoneArms`) and a child class that defines the implementation of that interface (`SkyStoneArmREV`). This organization is shown in Figure 2, below.

This organization is supported for several reasons, specifically, to separate the various robot components into logical groups, and to support multiple robots with potentially different hardware solutions to implementing its functions. The three components included in this group could have been separated further, but since they are small they were lumped together for convenience.

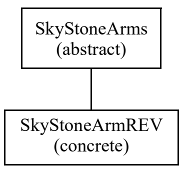

Figure 2 - Hook and Arm High-Level Structure

The hook is a single HiTech HB485 servo that operates a hooking mechanism used to latch onto the foundation in the building zone. It has three possible positions: latched onto the foundation, unlatched from the foundation, and parked for storage in a position that is out of the way. The latched position is called `Hook.hooked`, the unlatched position is called `Hook.open`, and the stored position is called `Hook.stored`. The hook is represented by a single function:

```
   public abstract void hook(Hook position);
```

The arm represents the function of raising and lowering the robot claw, which is used to grab the various stones on the playing field. It is able to raise and lower its position during the Driver Op Mode. The arm is also represented by a single function:

```
   public abstract void lift(double power);
```

The power argument sets the power to the arm, and must be in the range of ± 1. A positive value causes the arm to go up, while a negative value causes the arm to go down. The arm is implemented as a single 12 volt motor with a 40:1 gear reduction.

The final device, the claw, is also represented by a single function:

```
   public abstract void grab(boolean grab);
```

Its purpose is to latch onto Sky Stones and other stone objects on the playing field in order to maneuver them to the building zone and place them on the foundation. The two possible argument values are `true`, which means to close the claw or grab onto a stone, and `false`, which means to open the claw and release a stone. The claw, like the hook, is also implemented as a single HiTech HB485 servo.

Driver (Iterative) Op Mode

Our Driver Op Mode is implemented as a child class of the FTC class `OpMode`. It is an iterative Op Mode, which means it executes the main function (`loop`) every 1/10th of a second. If the `loop` function ever takes more than 4 seconds to complete its execution, the underlying software believes it has hung and aborts the program.

As a class descended from `OpMode`, it is required to implement five functions, namely, `init`, `init_loop`, `start`, `loop`, and `stop`. The function `init` is used to initialize the robot, e.g., find and initialize any hardware devices that might be needed. `init_loop` is executed repeatedly while the robot waits to start. Generally, it might be used to keep any devices warm that need to be, or it might be useful for executing any preliminary navigational functions that might be too slow to be included in `init`. So far we have not needed that particular function. The `start` function is used for the purpose of starting the game timer. The last two functions, `loop` and `stop`, are used during play. `loop` is used during the main portion of play, while `stop` is used to end play and shut down the robot.

Our Driver Op Mode is structured similar to our other components, that is, there is an abstract parent class (`SkyStoneDriver`) and a concrete child class (`SkyStoneDriverREV`). (See Fig. 3.) However, in this case the parent class provides more than abstract function declarations. Our Driver Op Mode provides implementations of those five functions in terms of our other abstract components, such as `SkyStoneArms` and `SkyStoneMotors`. Structuring the Op Mode in this way allows us to defer our choice of hardware until a later time. This makes it easy to support multiple robots with the same Op Mode, even when they use different hardware.

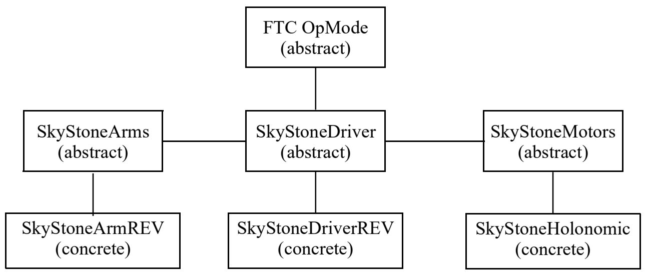
FTC OpMode (abstract)
SkyStoneArms (abstract)
SkyStoneDriver (abstract)
SkyStoneMotors (abstract)
SkyStoneArmREV (concrete)
SkyStoneDriverREV (concrete)
SkyStoneHolonomic (concrete)

Figure 3 - Driver Op Mode High-Level Structure

SkyStoneDriver

As mentioned before, the parent class of our Driver Op Mode is `SkyStoneDriver`. This class implements all of the abstract functions from the FTC class `OpMode`. These functions are written with only references to abstract components. For example, the class declares an object of type `SkyStoneMotors`, called `motors`. That object is not initialized by `SkyStoneDriver`, though. Instead, it is initialized by `SkyStoneDriverREV`, when its constructor is called. But `SkyStoneDriver` implements `loop` and other functions using references to `motors` under the assumption that it *has* been initialized and there is a concrete class it points to that properly implements its functions. Each of these abstract functions of the parent are then inherited by the child class `SkyStoneDriverREV`, or by any other child class that wishes to substitute some new hardware in place of that which was chosen.

Under this new approach it is the child class who is responsible for creating the component objects, such as `motors` and `arms` in its init function. This happens when the Op Mode is selected from the drop-down menu on the Driver Station. When the robot is initialized, it is the child's `init` function, and not that of the parent, that is called to create any needed devices for the robot.

A code snippet from the `loop` function is shown in Figure 4, and the complete `stop` function is shown in Figure 6. As you can see in both cases, the Op Mode is written using the abstract components in place of robot devices or the actual concrete objects. You can see this in Figure 4 on line 131 for the robot base, and lines 137 and 139 for the robot arms. In Figure 5 you can see this again on lines 180 and 181. Java ensures that abstract functions in the parent class have implementations in the child class, so using them in this fashion is guaranteed to be safe.

```
 69   public void loop() {
         ...
100      double bearing = 90 - alpha * 180 / Math.PI;
101
102   // figure out the driver's desired speed (make sure it is less than 1.0)
103      double length = Math.sqrt(x*x + y*y);
104      double speed  = Math.min(length, 1.0);
         ...
109      // get the turn rate from the triggers or bumpers
110      double turn   = 0;
111      boolean left  = (0<gamepad1.left_trigger ) || gamepad1.left_bumper;
112      boolean right = (0<gamepad1.right_trigger) || gamepad1.right_bumper;
113
114      if ((left && right) || (!left && !right)) {
115         ;           // do nothing
116      } else if (left) {
117         if (gamepad1.left_bumper) {
118            turn = slow_turn;
119         } else {
120            turn = fast_turn * gamepad1.left_trigger;
121         }
122      } else if (right) {
123         if (gamepad1.right_bumper) {
124            turn = - slow_turn;
125         } else {
126            turn = - fast_turn * gamepad1.right_trigger;
127         }
128      }
129
130      // send the driver commands to the motors
131      motors.move(bearing, speed, turn);
132
133      // open or close the claw
134      if (gamepad2.a && gamepad2.b) {
135         ;                   // do nothing
136      } else if (gamepad2.a){
137         arms.grab(false);   // open
138      } else if (gamepad2.b) {
139         arms.grab(true);    // close
140      } else {
141         ;                   // do nothing
142      }
         ...
172   }
```

Figure 4 - SkyStoneDriver loop Function

```
public void stop() {
   // stop the motors
   motors.stop();
   arms.stop();
}
```

Figure 5 - SkyStoneDriver stop Function

SkyStoneDriverREV

The concrete (non-abstract) child class is `SkyStoneDriverREV`. (The name ends with REV because this robot uses primarily REV Robotics hardware.) This is the class that assigns an implementation to the `motor` and `arm` objects. This is also where the actual Op Mode is created. The entire source code is short, and can be seen in Figure 6. In our example code, the implementation we use for our `motors` object is `SkyStoneHolonomic`. The implementation we use for our `arms` object is `SkyStoneArmREV`.

```
18 @TeleOp(name="SkyStone Driver REV", group="Nex+Gen Griffins")
19 public class SkyStoneDriverREV extends SkyStoneDriver {
20    private ElapsedTime runtime = new ElapsedTime();
21
22    @Override public void init() {
23    telemetry.addData("Status", "Starting Initialization.");
24
25       if (DeviceFinder.exists(this, "OwO")) {
26          motors = new SkyStoneHolonomic(this, 100, 6000.0 / 360.0, 5);
27       } else if (DeviceFinder.exists(this, "UwU")) {
28          motors = new SkyStoneHolonomic(this, 131, 6000.0 / 360.0, 5);
29       } else {
30          motors = new SkyStoneHolonomic(this, 120, 6000.0 / 360.0, 5);
31       }
32       motors.init();
33
34       arms = new SkyStoneArmREV(this);
35       arms.init(false);
36
37       // Tell the driver that initialization is complete.
38       telemetry.addData("Status", "Initialization Complete.");
39    }
40 }
```

Figure 6 - SkyStoneDriverREV Class

Once again, the beauty of this is that if we had an additional robot that used completely different hardware, we could support it by creating a new class for `motors` or for `arms`, as needed, and a new Op Mode class like this one, and there would be no need to re-implement all of the other code, where most of the work resides.

Notice also that on lines 25 - 31, this function is querying the robot for devices named “OwO” and “UwU”, using the boolean function `DeviceFinder.exists`. This is an example of a new technique we developed that allows us to identify at run-time which of several robots we are using, by creating a pseudo-device that matches the name of the robot. In this example, one of the parameters for the `SkyStoneHolonomic` constructor, that affects how many encoder clicks per inch the robot base uses, must be adjusted based on the robot. We accomplish this using our new technique by cycling through the names of each of our robots, and setting the limit when the robot we are using is found.

Autonomous (Linear) Op Mode

For this competition we have four separate Autonomous Op Modes, one for each quadrant of the playing field. The playing field is divided into two alliances, the Red Alliance and the Blue Alliance. The playing field is further divided into two zones, the loading zone and the building zone. The combination of alliance and zone defines the four quadrants: Red Loading Zone, Red Building Zone, Blue Loading Zone, and Blue Building Zone, each with its own requirements. Since both of our robots, OwO and UwU, use essentially the same hardware, there is no need to separate the Op Mode into an abstract parent class that defines the behaviors and concrete child classes that define the hardware, as we did for the Driver Op Mode. Instead, we explore a different technique, using class inheritance, to simplify our code and avoid code duplication.

The general structure of our Autonomous Op Mode can be seen in Figure 7. It uses the same components we used in our Driver Op Mode, but now we have four different Autonomous Op Modes, whereas we have only one Driver Op Mode. We have a different kind of code sharing that we need to worry about. Here we have code sharing over four different Op Modes, rather than sharing of an Op Mode over multiple robots. In fact, there is more code shared across Op Modes than there is code that is different between them.

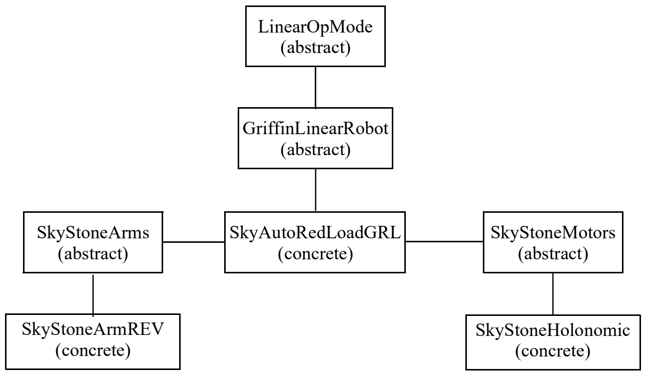
LinearOpMode (abstract)
GriffinLinearRobot (abstract)
SkyStoneArms (abstract)
SkyAutoRedLoadGRL (concrete)
SkyStoneMotors (abstract)
SkyStoneArmREV (concrete)
SkyStoneHolonomic (concrete)

Figure 7 - Autonomous Op Mode High-Level Structure

Although it is a bit of an advanced technique for new programmers, a standard approach in the industry would be to use inheritance to share the code. We collect all of the code that is common and place it within a new parent class, called `GriffinLinearRobot`. Each of the Autonomous Op Modes inherits from, or *extends*, in Java terminology, the `GriffinLinearRobot` class. This gives the each of the child classes all of the capabilities defined by the parent class.

We fully understand that this is not a new technique and we did not invent it. But we do wish to call it out as a matter of education for inexperienced programmers who might be reading this document, as a powerful technique that can help in some cases to create better code.

One more thing is worth mentioning. Our Autonomous Op Modes are all implemented as Linear Op Modes. This is different from our Driver Op Mode, which is implemented as an Iterative Op Mode. Linear Op Modes have only a single entry function, named `runOpMode`. Initializing, running and stopping the robot all take place as sections within that function. The main separator between initialization and running the op mode is a call to a function called `waitForStart`. In many ways, Linear Op Modes have a lot in common with regular programming. In particular, all commands are executed sequentially. Once the last command in the function is executed, the program terminates and the robot stops.

GriffinLinearRobot

This class contains four main functions, namely, `initialize_robot`, `execute_loop`, `shutdown`, and `exists`, for a little over 300 lines of code. There is no real common theme to this class, other than the fact that these four functions are shared identically across all four Autonomous Op Modes. The functions `initialize_robot` and `shutdown` should be fairly self explanatory, but the other two represent significant innovations and need to be explained further.

The `exists` function is not a lot of code, but it makes it possible to determine at run-time, and therefore make decisions based on, which of several robots you are using. This can be useful if your team supports more than one robot and the robots are not identical in some way. The function itself merely asks the hardware if a device with a given name exists, and catches the exception that is raised when the device doesn't exist in the robot configuration. Catching the exception is important to prevent the robot from aborting its execution on the error. The full listing can be seen in Figure 8.

```
public boolean exists(String name) {
   boolean result = false;

   if (name != null) {
      try {
         HardwareDevice device = hardwareMap.get(name);
         result = device != null;
      } catch (Exception e) {
         result = false;
      }
   }

   return result;
}
```

Figure 8 - SkyStoneDriver loop Function

The function becomes interesting when each robot has a device defined in the robot configuration, and the name of the device matches the name of the robot. One can cycle through the list of known robots, one at a time, until the one that is currently in use can be found.

There is one more detail, however. The FTC software also throws errors whenever a device named in the robot configuration does not have the correct device plugged in at the specified port. This is true for all device types except for one, Analog Input Devices. So, when the named device is created, it needs to either be backed up by hardware, or it needs to be an Analog Device.

The other function of interest in this class is `execute_loop`. This function implements an interpreter for the Griffin Robot Language, or GRL. An instruction in this language consists of one of five operations, namely, `GRAB`, `HOOK`, `MOVE`, `TURN`, and `SLEEP`, along with one or more parameters defined by that operation.

Instructions are represented as arrays of Java Objects, and programs are represented as arrays of instructions, that is, arrays of arrays of Java Objects. The interpreter simply loops over a program, executing one instruction at a time until all instructions have been executed. Each operation corresponds with one of the robot operations, and the parameters correspond with the arguments to that operation. That is the concept.

The full language is given in Figure 9, below. A short example is given in Figure 10.

```
GRAB, grab/release, message
HOOK, open/close, message
MOVE, bearing, power, range, message
TURN, bearing, power, message
SLEEP, milliseconds, message
```

Figure 9 - Griffin Robot Language

```
Object[][] owo_instructions = {
   { SLEEP, 20000,           "sleep for 20 seconds"},
   { MOVE,     90, 0.50, 12, "move 12 in. to the right"},
};
```

Figure 10 - Example Griffin Robot Language Program

This simple program sleeps for 20 seconds, then moves 12 inches to the right. The parameter to the sleep instruction is the amount of time to sleep, in milliseconds. The parameters to the move instruction are the direction in degrees, the power to use, and the distance in inches to move. The final parameter in both cases is a message to be sent to the Driver Station as the instruction is being executed.

There are several advantages to using this language. A robot Op Mode is represented much more compactly than before, which makes it more easily understood and modified, and it is stored as a data structure in memory, making it possible to modify the robot program by simply changing the value of a pointer. The advantages may be subtle, but in the right place they are also powerful.

There is really only one trick to implementing this interpreter, and that is how you convert a Java Object back into parameter of a data type you can use. For this you need to use a type cast, but parameters not necessarily stored the way you expect, so you have to expect the unexpected. Casting an Object as a `float` that is stored as an `int`, for example, results in an exception that has to be handled or the program will terminate prematurely. So, each cast is enclosed within a try/catch block, and when an exception is thrown, a different type cast is tried. These techniques are codified as the private functions `to_int`, `to_double` and `to_long`.

SkyAutoRedLoadGRL and Other Op Modes

We are now ready to see a complete listing of the `runOpMode` function. (See Fig. 11.) The only thing missing from this is the list of instructions for the robot to execute, like our example in Figure 10.

```
@Override public void runOpMode() {
   telemetry.addData("runOpMode", "starting init"); telemetry.update();
   initialize_robot(false);
   telemetry.addData("runOpMode", "init complete"); telemetry.update();

   Object[][] instructions = null;

   if (exists("OwO")) {          // is this robot OwO?
      instructions = owo_instructions;
   } else if (exists("UwU")) {   // is this robot UwU?
      instructions = uwu_instructions;
   } else {                      // we don’t know, use OwO
      instructions = owo_instructions;
   }

   waitForStart();

   if (instructions != null) {
      execute_loop(instructions);
   }

   shutdown();
}
```

Figure 11 - Red Alliance Loading Zone Autonomous Program

Test (Linear) Op Modes

There are two remaining Op Modes that are used for calibrating the robot. The first, `SkyAutoMoveGRL`, moves forward 10 feet and stops. The second, `SkyAutoTurnGRL`, turns 360 degrees to the right and stops.

Team Software Documentation and Source Code Repository

All team software and documentation, including Relic Recovery, Rover Ruckus and Sky Stone, is stored and maintained within a git repository, and is publicly available at:

https://github.com/dmpase/griffin.robotics

Code and documentation for the SkyStone branch specifically can be found at:

https://github.com/dmpase/griffin.robotics/tree/SkyStone

Questions on any topic can be directed at any time, to our team or to our mentor, at:

dmpase@gmail.com