nex+Gen Griffin Robot Software Innovations

Griffin Robotics, Team 7582

General Approach

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. Other teams may share these goals as well, but they make the task of designing and implementing the Op Modes more challenging. We list these additional goals, and their corresponding innovations, below. We also describe the goals, and how they were achieved, in later sections.

Software Architecture Goals

1. Easy for new team members to become effective programmers on the robot
2. High-level operations for robot control (e.g., move forward, turn, open claw, etc.)
3. Reusable components from year to year (e.g., holonomic transportation base)
4. Re-usable components across multiple Autonomous Op Modes
5. Mechanisms for handling large-scale robot differences (e.g., using servos vs. 12v motors)
6. Mechanisms for handling small-scale robot differences (e.g., Mecanum vs. Omni wheels)
7. Mechanisms for handling very small-scale differences in robot behavior (described later)

Innovations to Meet These Goals

1. An abstract class hierarchy that makes programming the robot easy to get started for new programmers, easy to use for experienced programmers, easy to reuse components from one year to the next, and supports multiple robots of different design within the same year
2. A way to share common code across Autonomous Op Modes in a common parent class
3. A mechanism to identify different robots by creating a fake device in the robot configuration
4. A simple, high-level robot control language to simplify Autonomous Op Modes

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

General code and documentation is found by looking in the *master* branch, while documentation and code specific to a competition can be found in the *Relic Recovery*, *Rover Ruckus*, and *Sky Stone* branches. Select the desired branch using the Branch drop-down box on the left of the page. For SkyStone specifically, you may also use the URL:

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

Team Software Architecture Goals

Goal 1: Easy for new members to learn

At the start of each competition we have a large influx of new members joining the team, and at the end of a season we lose a significant amount of experienced members because they graduate or move on to other interests. This constant turn-over of team membership, combined with our choice of using the Java Programming Language, makes it difficult to maintain a pool of competence within our programming staff. So, our first goal has been to figure out how we could make programming the robot more accessible to inexperienced programmers.

We accomplished this goal, and the next, by creating a set of simple, intuitive, high-level operations that new programmers could easily relate to, like move the robot forward, or turn the robot to one side.

Goal 2: High-level operations for robot control

A challenge we identified early on is that the FTC software base, at least for Java, provides ample ability to control and monitor many different types of hardware devices, but programming the robot to perform high-level tasks, like those in Autonomous Op Modes, needs the programmer to also think in terms of high-level, coordinated actions and not just about individual devices. An example of a high-level operation might be to move the robot forward 12", or turn left 30 degrees. Each of these actions requires the programmer to use several devices at once in a coordinated way. Without these high-level actions, even simple tasks can become overwhelmingly complex.

In order to keep the robot design simple, we created separate abstract layers for each of the major subsystems. Our subsystems include the travel base, robot arm and hook, etc. Each abstract layer is a set of functions that represents what the subsystem can do. For example, the travel base can move in different directions, so there is a `move` function. The claw can grab a stone, so there is a `grab` function. In all we identified seven main high-level operations to manage all of the operations of the robot. They are: `move`, `move_to`, `turn_to`, `lift`, `grab` and `hook`. The details of *how* each action is accomplished are specific to the hardware, and as such are handled separately.

The benefit may appear subtle, but programming both Driver and Autonomous Op Modes were hugely simplified by defining a set of reusable abstractions. It is an application of the programming principle of *divide-and-conquer.*

An experienced programmer might naively conclude that this is nothing more than collecting operations into methods. We do use class methods, but it is more than just that. We structure the robot components as abstract Java classes in order to achieve our other goals. *Please read on!*

Goal 3: Reusable components from one year to the next

The team also wanted to learn from the best practices we identified in previous competitions. Each year we develop a significant amount of software and we wanted to be able to reuse software from one year to the next. Part of the exercise of engineering is to learn what works best and incorporate it into the team product or processes going forward. Engineering is far less effective if the team has to relearn all of the old lessons from one year to the next. Our software

structure creates a collection of flexible, reusable software components that can be reused from one year to the next as a major part of our architecture.

We started this goal last year, and that is when we defined the reusable modules. This year was our first year of really testing whether we got it right. Our experience this year shows that we have been completely successful. Although the Op Modes are specific to the competition, such as SkyStone or Rover Ruckus, we were able to reuse all of last year's other components with very few modifications.

Goal 4: Reusable components across multiple Op Modes

Another problem that arose was that, even with a single robot, we have multiple Autonomous Op Modes. Because of the way the game is structured, we often end up with a separate Op Mode for each quadrant in the playing field. This creates the potential for a substantial amount of duplicate code. The challenge of duplication is that different copies can easily get out of sync, that is, become different from each other, unless a lot of extra care and effort is made.

We solved this problem by pulling out common components from the multiple Autonomous Op Modes and incorporating them into a single parent class. Code that was common included robot initialization, shut-down, and other utility routines. This represents a relatively small fraction of the total code, only around 200 lines of code, but remember the problem is *differences* and not just *size*. By moving common code to a parent class that is shared by all Op Modes, only a single copy of this software has to be created and maintained.

Goal 5: Mechanisms for handling large-scale differences between robots

Next, the team had a desire to support multiple robots. With multiple robots they can experiment with different ideas to see what works best under different circumstances. Furthermore, if the team grows large enough, they can support multiple teams with the same code base, even when the teams design different robots. To do this, the code structure has to have the flexibility to handle large-scale difference between robots. One example of this was during Relic Recovery, when our Lion team designed a claw with two servos and our Eagle team designed the same component with a single 12-volt motor.

The tricky bit is that it has to be done without duplicating those things that *don't* change, like the high-level operations the robot performs during the autonomous portion of the competition. In our Lions and Eagles example, both teams used the same general operations (`move`, `move_to`, etc.), so their Op Modes could share code[1]. They also used the same holonomic base, so they could use the same code for the base.

The mechanism we used to solve this problem turned out to be the same one we used to make the robot program easy to start learning, easy to use, and reusable from one year to the next. We divided the software into a hierarchy of abstract and concrete classes, which made the code easy to share between robots, but the hierarchy also made it easier to learn, to use, and to reuse. We will describe this hierarchy in more detail in a later section.

1. Our experience with Relic Recovery, described in this example, was the inspiration for creating this program structure. Code sharing across robots was more difficult until we moved to this solution.

Goal 6: Mechanisms for handling small-scale differences between robots

We recently came up with an innovation to solve yet another vexing problem. This year we are supporting two robots that are almost, but not quite, identical. This leads to a situation where the differences between robots are small enough that it requires only minor tweaks between them. For example, we have experimented with slightly different wheels -- Omni and Mecanum -- in our robot base. The operation of the two is nearly identical, they both support holonomic operation, *but the wheels are different sizes*, and therefore require a different number of wheel rotations to move a certain distance. We rely on dead reckoning to navigate the playing field, so differences in wheel sizes have to be addressed or the robot will not work properly.

The mechanism we described in the previous section, creating a new Op Mode with reusable components, is certainly capable of supporting minor differences like this. Unfortunately, that requires a bit of extra effort and it would be more convenient if there were a simple mechanism we could use in short code snippets. Creating a reusable component involves creating separate classes to encapsulate the differences, but all we really need is to be able to tweak a few parameters on a couple of lines.

To solve this problem we invented a technique that accomplishes this by creating a fake device in the robot configuration that identifies which robot is running. The program asks the robot whether a device exists that matches the name of the robot, which comes back true or false. In this way the program asks about different robots by name until it finds one that matches, and sets the parameters accordingly.

Goal 7: Mechanisms for handling very small-scale differences between robots

In supporting multiple robots we uncovered a third kind of difference that can occur, one that is neither large scale, like using different hardware components to accomplish the same function, nor is it parametric, like using different sized wheels. We found that under certain conditions the robots exhibited differences that weren't easily addressed by our other two mechanisms. Our specific example is during one leg of one of the Autonomous Op Modes, our robot that uses Mecanum wheels moves a little differently than the other robot, that uses Omni wheels.

The problem is that it is *only* during one leg of one Autonomous Op Mode, and it is consistent. We addressed most of the differences, like using different wheel designs, by using fake devices to identify the robot, but there is something about the distribution of weight in the robot, placement of wheels, or some small detail like that, that isn't taken care of by our other solutions.

We solve this problem with a solution that might sound odd: we design a high-level robot programming language and convert the Autonomous Op Modes into streams of instructions in that language. This solves the problem because we can represent each Op Mode as a compact and easily understood sequence of robot commands stored as a Java array. Each robot has its own sequence of commands, which allows corrections to be used where they are needed. The concern is that we now have duplicate code, but that is mitigated by the fact that the instruction streams are simple, intuitive and both fit together within a single page, and that allows them to be easily compared for undesirable differences.

General Solutions to Implementing Our Goals

We discovered that four separate techniques, if used in our software architecture, would meet all of the goals we set out to achieve. These techniques, which goals they satisfy, and how they are implemented, are described below.

Software Architecture Techniques

1. By using a combination of Java abstraction and inheritance in a specific way, described below, we were able to divide the robot into easy to learn, easy to use components that we can re-use each year.
2. By separating code that is common across all Autonomous Op Modes into a single parent class, we are able to avoid a significant amount of code duplication.
3. By creating a fake device and using the Java technique of catching exceptions, we are able to identify which robot we are executing in, and make adjustments to important code parameters.
4. By creating a compact, easily understood robot programming language, we allow for minor changes to Autonomous Op Modes between robots without sacrificing program clarity

Technique 1: Using abstraction and inheritance to structure our code

We discovered that if we structured our code in a certain way, we could meet our goals of (1) low effort to get started, (2) high-level robot operations, (3) reusable components year-over-year, and (5) mechanisms for handling large-scale differences between robots. All of these goals are met with the same general technique, but the technique is subtle and requires some explanation. Our Driver Op Mode structure is illustrated in the next diagram. (See Fig. 1.)

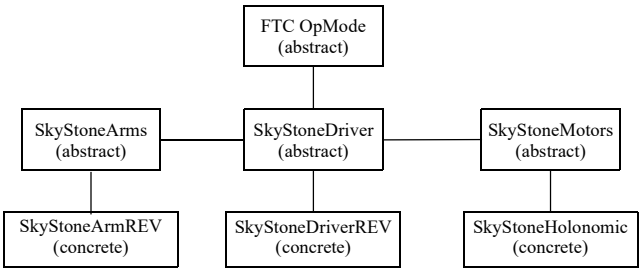

Figure 1 - Driver Op Mode High-Level Structure

To achieve these different goals requires that we split each of the robot subsystems into two layers: an interface layer and an implementation layer. The interface layer creates an abstract declaration of *what the robot does*, while the implementation layer creates a concrete definition of *how the robot does it*.

So, the way this works is the abstract layer defines what operations the subsystem is able to perform. For example, the `SkyStoneMotors` class defines the operations the travel base can do, e.g., move in some direction at a certain speed, move a given distance, turn some number of degrees, turn at a certain speed. These operations are given as Java abstract functions. As an example, the operation “move at a certain speed” is the function:

```
public abstract void move(double bearing, double power)
```

By declaring it as an abstract function, an Op Mode like `SkyStoneDriver`, that uses an object of type `SkyStoneMotors`, is free to use all of the abstract methods when it creates the body of the Op Mode, without specifying *which* specific travel base it is using. But, if it chooses *not* to specify the base, it becomes an abstract class itself, and in order to use the Op Mode there must also be a concrete class derived from it that *does* specify which concrete motor base used. In this example, the concrete class is `SkyStoneDriverREV`, which does spell out which concrete classes it uses.

How does this help? By splitting the Driver or Autonomous Op Mode in this way, we have a single abstract Op Mode that defines what the robot does in terms of these abstract operations, like `move`, that are defined by the subsystem abstract parent class, like `SkyStoneMotors`. One year, for the Rover Ruckus competition, we had two different robots, each with their own arm and claw assembly to move the minerals around. We implemented a single abstract Driver Op Mode and a single abstract arm class, but we had two concrete implementations of the arms, one for each design, and two concrete Driver Op Modes. The abstract Driver Op Mode, where most of the real work is done, was the same for both robots. The concrete arm implementation was different, of course, and there were two short concrete Driver Op Modes, whose only task was to inherit the abstract Op Mode and to declare which concrete components were being used. Changes to the *behavior* of the Op Mode took place in the abstract Driver Op. Changes to the *implementation* of the robot subsystems -- arm or claw or travel base -- took place in the concrete child classes.

Using this structure makes the robot a little more complex in some ways, but notice all the benefits that come from it. By creating an abstract layer that describes the operations the robot is able to do, even beginning programmers can become effective with just a small amount of training. Both Driver and Autonomous Op Modes are more intuitive and much simpler to program. Abstract and concrete components become reusable from one year to the next. This year we reused the abstract and concrete layers from our travel base and arm subsystems with only a few modifications. And last but not least, we can support multiple *different* robots in the same year with the same code base.

Technique 2: Separate code that is common across all Autonomous Op Modes into a single parent class

We discovered that by using inheritance, one of the most important features of Java (and indeed, *all* object-oriented languages), we could accomplish goal (4) and reuse common code across our Autonomous Op Modes.

Our Autonomous Op Modes are structured as shown in the next diagram. (See Fig. 2.) We have four separate Op Modes, one for each quadrant. As we developed these Op Modes we initially developed one quadrant, then modified copies for each of the remaining quadrants. As we expanded and refined these Op Modes we discovered that some of the code that should have been the same, was different across the Op Modes, and that difference was affecting our ability to create proper solutions.

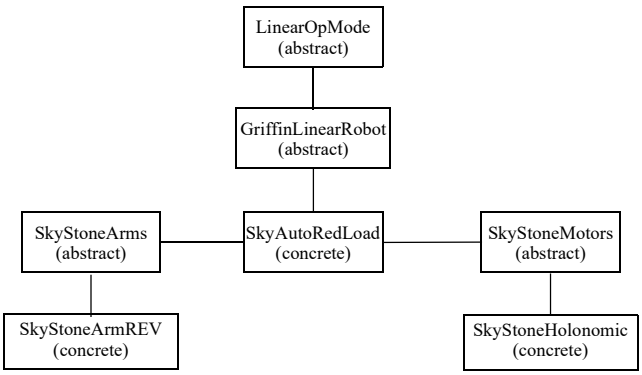

Figure 2 - Autonomous Op Mode High-Level Structure

The astute programmer will recognize that the solution to this problem is just what we've done here, that is, collect all common code into a single parent class and allow the different Op Modes to inherit that class to gain access to the code. The FTC software uses inheritance extensively as a mechanism for sharing features, and that makes it necessary to do so here.

Technique 3: Using a fake device in the robot configuration to identify which robot is running

As we described in an earlier section, we invented a technique that identifies which robot is running by creating a fake device in the robot configuration. The program asks the robot whether a device that matches the name of the robot exists within the configuration. The query comes back as either true or false, depending on whether the device exists. In this way the program asks about different robots by name until it finds one that it recognizes, and sets any relevant program parameters accordingly.

The challenge with this approach was doing it in a way that does not cause the robot program to abort. We solved *that* problem with the help of two observations. First, we noticed that the FTC software throws a Java Exception whenever a program requests a device that is not registered in the robot configuration. Second, we also noticed that exceptions are thrown even when the device is registered if the device is not plugged in correctly, *except for analog devices.* We used these two observations to formulate a solution in three steps.

For the first step we need to attach a name to the robot in a way that software can find it. To do this we define an analog device in the robot configuration whose name matches that of the robot. It does not matter which port the name is attached to, as long as it is an analog device. No physical device needs to be physically attached to the REV Expansion Hub at that port, either, and it's simpler if none exists.

Next the software must look-up the name in the robot configuration. This requires a function that is able to handle failures when the FTC software does not find the requested name. Fortunately, this can be done using another key feature of Java, exception handling. The look-up function follows the usual device look-up procedure, but encapsulates it in a try-catch block, as below.

```
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 3 - Look-Up Routine to Identify a Robot

This function, or class method, has to be part of a class that inherits from the FTC class `OpMode` or class `LinearOpMode`, but that includes all Autonomous and Driver Op Modes.

The final step is that the Op Mode must call this function on every robot name that it wishes to identify until it finds one that matches.

Technique 4: Create a simple, high-level robot programming language for Autonomous Op Modes

This is actually easier than it sounds. All we need is to have some way of representing a sequence of instructions and parameters for those instructions, and Java gives us those mechanisms as part of the native language. Once we have decided on the form of the language, we need a way to translate robot instructions into calls to our high-level methods.

The easiest way to represent an instruction is by using an array of Java Objects. Each instruction needs an operation code, or an *op code*, and whatever values are needed to execute that operation. For example, one of the operations might be to *grab a stone*. Another operation might be to *release a stone*. These could be implemented as two separate instructions or as a single instruction with an argument that indicates whether to grab or release the stone. An instruction is represented by an Object array, where the first element of the array is always the op code. All subsequent elements of the array are defined by the op code. Our full language is given below:

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

Figure 4 - High-Level Robot Programming Language

In this language, `GRAB`, `HOOK`, `MOVE`, `TURN`, and `SLEEP` are Java symbols we have defined within the program.

For `GRAB`, the value `grab/release` is a boolean value, either `true` or `false`, where `true` means to grab a stone and `false` means to release one. The value `message` is an optional Java `String` that is sent to the driver.

For `HOOK`, the value `open/close` is also a boolean value, where `true` means to close the hook and `false` means to open it. As before, the message is an optional Java `String`, and when included, it is sent to the driver.

For `MOVE`, the value `bearing` is a double-precision floating-point value that represents the direction the robot will take. The direction is an angle, in degrees, where 0 degrees is forward, 90 degrees is to the right, 180 degrees is directly behind, and both -90 degrees and 270 degrees is to the left. The base is holonomic, so the robot can travel at any angle between -360 and +360 degrees. The robot keeps its original orientation, that is, it does not turn towards the new direction. The value `power` is also a double-precision floating-point number that represents the power to be applied to the wheels. Power must be a value between -1.0 and +1.0. The value `range` is another double-precision floating-point value, and it represents the distance, in inches, the robot must travel. As before, the message is optional, and when included, it is sent to the driver.

For `TURN`, as for `MOVE`, the value `bearing` is a double-precision floating-point value that represents the direction the robot will turn. The direction is an angle, in degrees, where 0 degrees is forward, 90 degrees is to the right, 180 degrees is directly behind, and both -90 degrees and 270 degrees is to the left. The value `power` is also a double-precision floating-point number that represents the power to be applied to the wheels. Power must be a value between -1.0 and +1.0. As before, the message is optional, and when included, it is sent to the driver.

For `SLEEP`, the value `milliseconds` is a 64-bit integer that represents the amount of time the robot will sleep before executing its next instruction. As before, the message is an optional Java `String`, and when included, it is sent to the driver.

A complete program for the Red Alliance loading zone, including the Java code for the `runOpMode` that drives it, is shown for the robot named OwO in Figure 5. From this example, it is easy to see how simple programming the robot can really be.

The final component for this feature is the interpreter for this language. If that component is too large or complex, this approach would not be feasible. Our implementation requires about 70 lines of fairly simple code. We show the code in its entirety in Figure 6 to help you see what it requires. The interpreter is called `execute_loop`, and is part of the `GriffinLinearRobot` class.

Our SkyStone code is available in our GitHub repository at:

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

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

dmpase@gmail.com

```
private static final double power                 = 0.75;
private static final double foundation_turn_power = 0.50;
private static final double robot_width = 16;
private static final double robot_depth = 14;

Object[][] owo_instructions = {
   {GRAB,  false,                                "open claw"},
   {HOOK,  false,                                "open hook"},
   {MOVE,  0,     power,48 - robot_depth + 2,    "move to stones"},
   {GRAB, true,                                  "grab a stone"},
   {SLEEP, sleep_delay,                          "sleep for claw"},
   {MOVE,  180,   power,                    14,  "back away from stones"},
   {MOVE,   90,   power,                    42,  "move to bridge"},
   {GRAB,false,                                  "drop stone"},
   {TURN,   -5,   foundation_turn_power,         "reorient robot"},
   {MOVE,   80,   power,36 + robot_width*3/4,    "move to foundation"},
   {MOVE,    0,   power,                     9,  "move forward"},
   {HOOK,  true,                                 "close hook"},
   {SLEEP, sleep_delay,                          "sleep for hook"},
   {MOVE,  180,   power, 12,                     "pull the foundation back"},
   {TURN, -225,   foundation_turn_power,         "turn the foundation"},
   {GRAB,false,                                  "open claw"},
   {MOVE,    0,   power,                    36,  "push the foundation"},
   {MOVE,  180,   power,                    12,  "back away from foundation"},
   {MOVE,   90,   power,                    54,  "move to bridge"},
};

@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 5 - Red Alliance Loading Zone Autonomous Program

```
public void execute_loop(Object[][] instructions) {
    if (instructions == null || ! opModeIsActive()) {
        // something is coded incorrectly
        shutdown();
        return;
    }

    try {
        for (Object[] instruction: instructions) {
            if (instruction == null || ! opModeIsActive()) {
                // player hit the stop button
                shutdown();
                return;
            }

            int op_code = (int) instruction[OP_CODE];
            if (op_code == GRAB) {
                // execute a grab operation
                boolean grab = (boolean) instruction[CLOSE];
                String message = (String) instruction[GRAB_MSG];
                telemetry.addData("execute loop", message);
                telemetry.update();
                arms.grab(grab);
            } else if (op_code == HOOK) {
                // execute a hook operation
                boolean close = (boolean) instruction[CLOSE];
                String message = (String) instruction[HOOK_MSG];
                telemetry.addData("execute loop", message);
                telemetry.update();
                if (close) {
                    arms.hook(SkyStoneArms.Hook.hooked);
                } else {
                    arms.hook(SkyStoneArms.Hook.open);
                }
            } else if (op_code == MOVE) {
                // execute a move operation
                double bearing = (double) instruction[BEARING];
                double range   = (double) instruction[RANGE];
                double power   = (double) instruction[POWER];
                String message = (String) instruction[MOVE_MSG];
                telemetry.addData("execute loop", message);
                telemetry.update();
                motors.move_to(bearing, range, power);
            } else if (op_code == TURN) {
                // execute a turn operation
                double bearing = (double) instruction[BEARING];
                double power   = (double) instruction[POWER];
                String message = (String) instruction[TURN_MSG];
                telemetry.addData("execute loop", message);
                telemetry.update();
                motors.turn_to(bearing, power);
            } else if (op_code == SLEEP) {
                // execute a sleep operation
                long sleep_time = (long) instruction[TIME];
                String message = (String) instruction[SLEEP_MSG];
                telemetry.addData("execute loop", message);
                telemetry.update();
                sleep(sleep_time);
            }
        }
    } catch (Exception e) {
        // something is coded incorrectly
        shutdown();
        return;
    }
}
```

Figure 6 - Interpreter for the High-Level Programming Language