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 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*, which are 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 this component is shown in Figure 1.

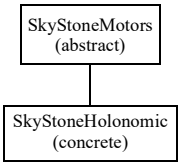

Figure 1 - Robot Base High-Level Structure

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` fuctions are used during the Autonomous Op Mode 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 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:

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

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, that has Mecanum wheels, seems to do best with a ratio of 128 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 mutiple 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.

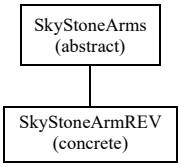

Figure 2 - Hook and Arm High-Level Structure

The hook is a single 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 hook is implemented as a single HiTech HB485 servo.

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.

Autonomous (Linear) Op Mode

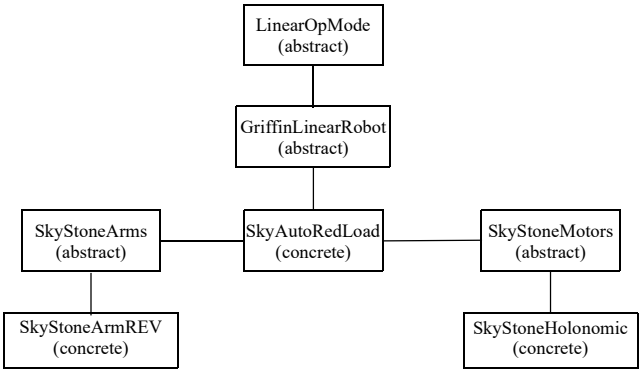

Figure 2 - Autonomous Op Mode High-Level Structure

Driver (Iterative) Op Mode

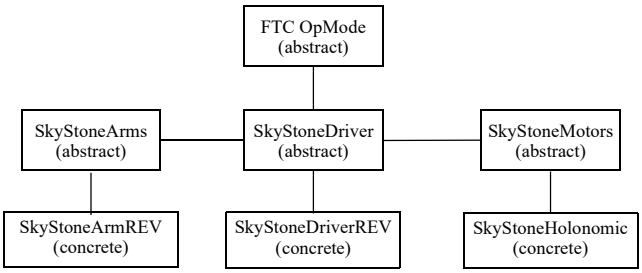

Figure 1 - Driver Op Mode High-Level Structure

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.

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

dmpase@gmail.com