FTC Java Programming Topics

Nex+Gen Griffin Robotics FTC Team 7582

Charles Stallings, Violet Frazier, Cameron Pase
Douglas Pase, dmpase@gmail.com

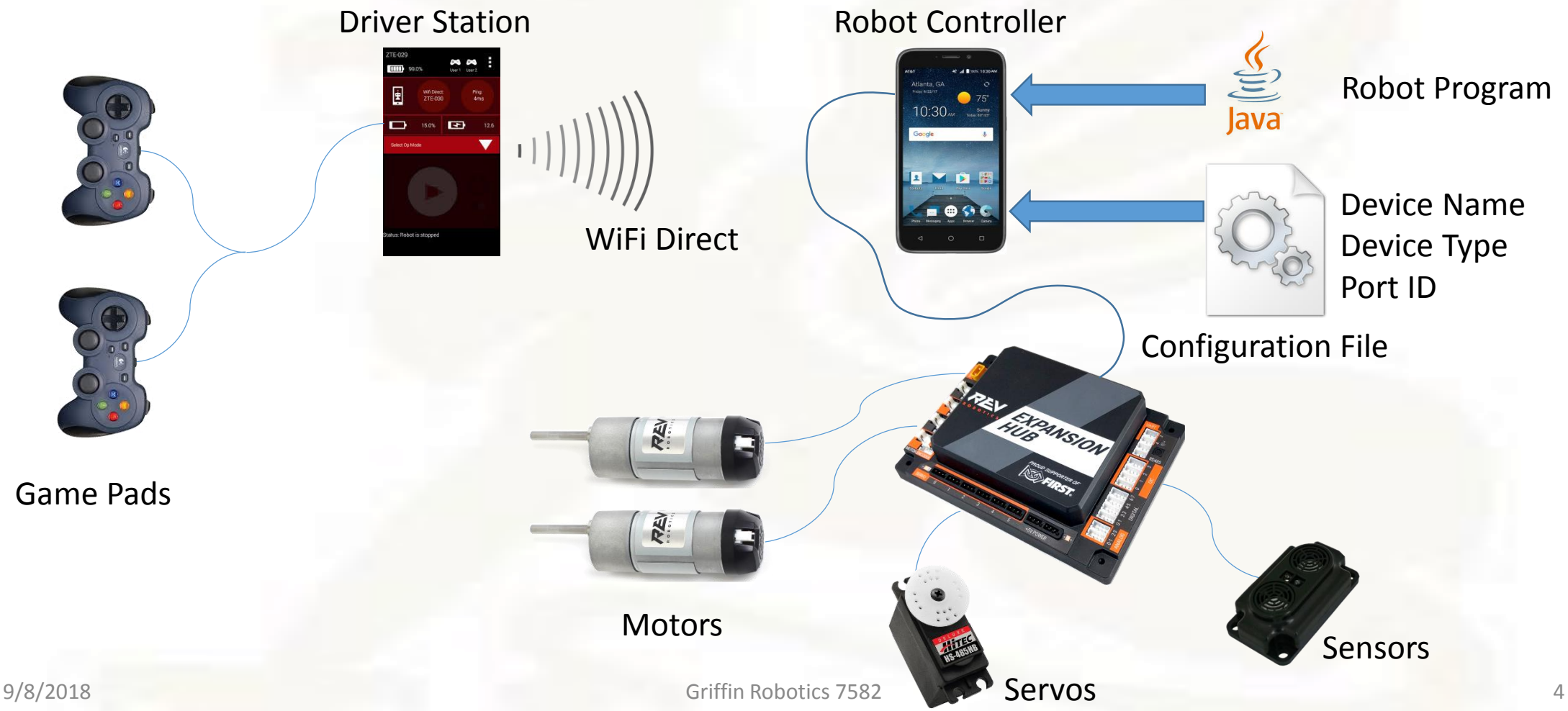
Agenda

1. 1:00 – 1:55: Robot Programming Basics
   - Linear (Autonomous) Programming Model
   - Iterative (Driver) Programming Model
   - Talking To The Robot
2. 2:00 – 2:55: Advanced Programming
   - Using Inheritance And Abstraction To Support Multiple Robots
   - Using Exceptions To Handle Hardware Failures
3. 3:00 – 3:55: Special Topics
   - Open Discussion And Q/A

Agenda

1. 1:00 – 1:55: Robot Programming Basics
   - Linear (Autonomous) Programming Model
   - Iterative (Driver) Programming Model
   - Talking To The Robot
2. 2:00 – 2:55: Advanced Programming
   - Using Inheritance And Abstraction To Support Multiple Robots
   - Using Exceptions To Handle Hardware Failures
3. 3:00 – 3:55: Special Topics
   - Open Discussion And Q/A

Robot Architecture



Linear Programming Model

- Easiest to understand
- Most like a typical Java application
  - Sequential execution from start to finish – do task A, task B, task C, …
  - When the last task is done, the robot stops
- Best suited for autonomous operation
  - Autonomous mode executes tasks in sequence, just like Linear Model
- Phases are not built into the model
  - Init, wait, main and stop operations must be explicitly coded into the program

Linear Code

```
@Autonomous(name="Autonomous Mode", group="Autonomous")
public class Autonomous_Linear extends LinearOpMode {
    private ElapsedTime runtime = new ElapsedTime();
    @Override
    public void runOpMode() {

        …

        waitForStart();


        runtime.reset();

        if (opModeIsActive()) {
            …
        }

         …
    }
}
```

```
// name used on driver station
// superclass must be LinearOpMode
// timer object
// java safety feature
// required, your robot code

// initialization section

// wait for start to be pressed

// main section, robot does work
// reset robot clock

// check if “stop” was signaled
// do next leg of operation
// …

// stop section, shut down robot
```

Initialization Section

- `runOpMode()` is called when “Init” is pressed on the Driver Station
- Immediately begins with the initialization of the robot
- Turn on all devices
- Each device (motor, servo, sensor) must be activated individually
- Activating a device places it into a known state, ready for use
- Extends from the start of the `runOpMode()` function to `waitForStart()`

Wait Section

- Waits for the operator to press “Play”  on the Driver Station
- Implemented on the robot with the call to `waitForStart()`
- When `waitForStart()` returns, “Play”  has been pressed

Main Section

- Begins immediately following `waitForStart()`
- Performs the real work of your autonomous mode
- Robot timer should be restarted at the beginning
- Each leg should check whether “Stop”  has been pressed
- When `opModeIsActive()` returns false, “Stop” has been pressed

Stop Section

- No distinct marker separating “Main” and “Stop” sections
- Begins after the last leg of the “Main” section, or whenever the real work is complete
- Shuts-down the robot components (e.g., motors, servos, sensors)
- When the `runOpMode()` routine exits, the robot stops

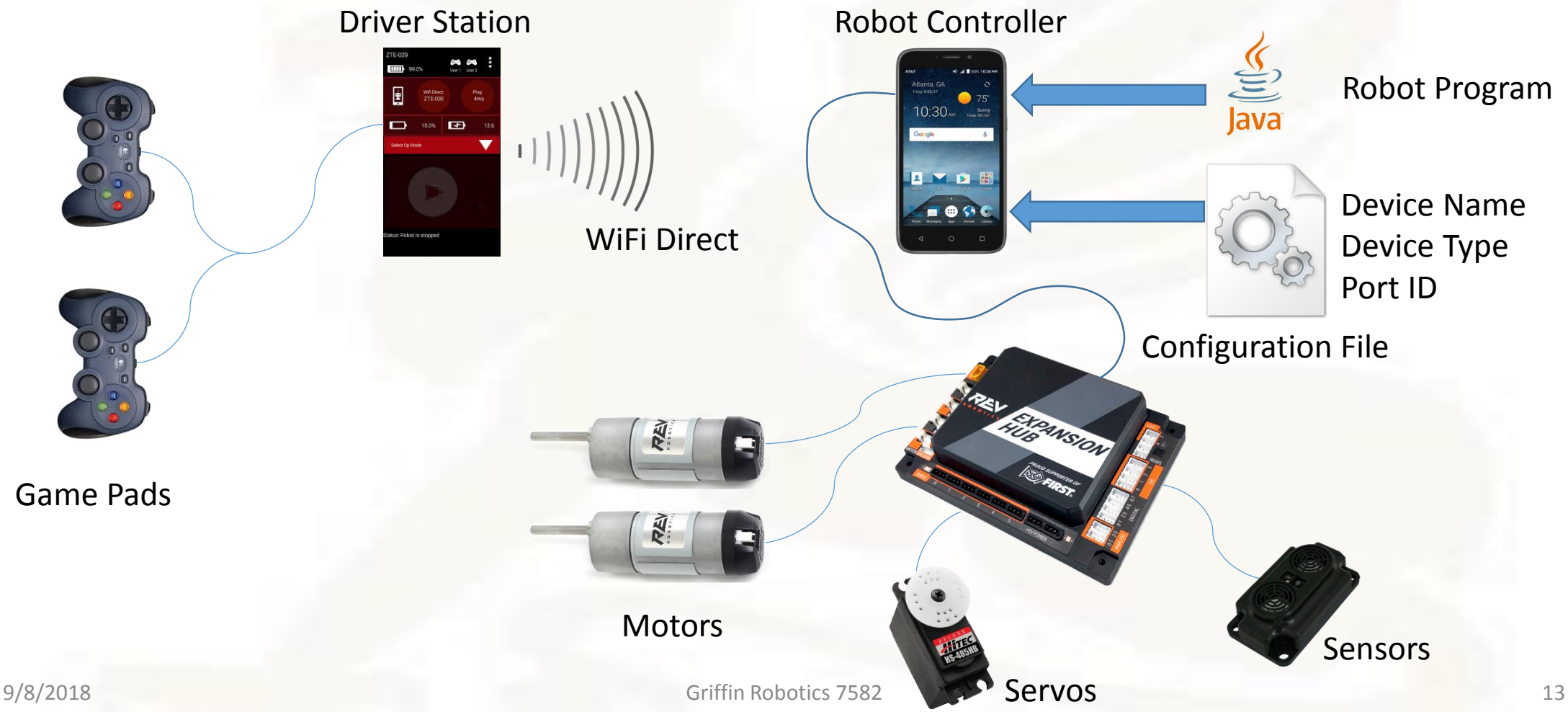
Telemetry

- Telemetry sends data from the robot to the driver station for display
- The telemetry object is inherited from OpMode or LinearOpMode
- `telemetry.addData(String caption, String value)`
- `telemetry.addData(String caption, String format, args…)`
- `telemetry.update()`
  - In Linear Op Mode, update is needed to display data on the driver station

Agenda

1. 1:00 – 1:55: Robot Programming Basics
   - Linear (Autonomous) Programming Model
   - Iterative (Driver) Programming Model
   - Talking To The Robot
2. 2:00 – 2:55: Advanced Programming
   - Using Inheritance And Abstraction To Support Multiple Robots
   - Using Exceptions To Handle Hardware Failures
3. 3:00 – 3:55: Special Topics
   - Open Discussion And Q/A

Robot Architecture



Iterative Programming Model

- Most suitable for driver operation
- Separates the operation into “init”, “loop” and “stop” sections
  - Two additional sections, “init_loop” and “start”, are also available
- Sections are implemented as individual routines
- Unused routines may be inherited from the OpMode superclass
- Uses “polling” for the main (loop) phase (explained later)

Iterative Code

```
@TeleOp(name="Driver Mode", group="Iterative Opmode")        // name used on driver station
public class Driver_Iterative extends OpMode {               // superclass must be OpMode
    private ElapsedTime runtime = new ElapsedTime();         // timer object
    @Override
    public void init() {                                     // initialization section
        …
    }
    @Override
    public void init_loop() { }                              // initialization loop section
    @Override
    public void start() { runtime.reset(); }                 // start section
    @Override
    public void loop() {                                     // loop section
        …
    }
    @Override
    public void stop() {                                     // stop section
        …
    }
}
```

Initialization Section

- Called once after “Init” is pressed on the Driver Station
- Initialization code is placed in the `init()` routine
- Turn on all devices in the robot
- Each device (motor, servo, sensor) must be activated individually
- Activating a device places it into a known state, ready for use
- Works similar to the initialization section of a Linear Op Mode
- Times out if it does not finish in less than 4 seconds
- Timing out aborts the program and crashes the robot

Init_loop

- Executes repeatedly after `init()` and before `start()`
- Useful for keeping devices “warm” between initialization and play
- Code is placed in the `init_loop()` routine
- Not often used or needed
- May also time out if not completed quickly

Start

- Executes once after the driver presses “Play”  on the driver station
- Used to reset the robot clock at the beginning of play
- May also time out if not completed quickly

Loop

- Executes repeatedly after “Play”  is pressed
- Continues to execute until “Stop”  is pressed
- Normal operation often follows this pattern:
  1. Read input from the game pads, sensors, and encoders
  2. Compute changes to the motor and servo power settings
  3. Set the new power levels for the motors and servos
- Executed about every 1/10th second (polling)
- Times out if it does not finish in less than 4 seconds
- Timing out aborts the program and crashes the robot

Stop

- Executes once after the “Stop” button  has been pressed
- Shuts-down the robot components (e.g., motors, servos, sensors)
- When `stop()` exits, the robot program terminates

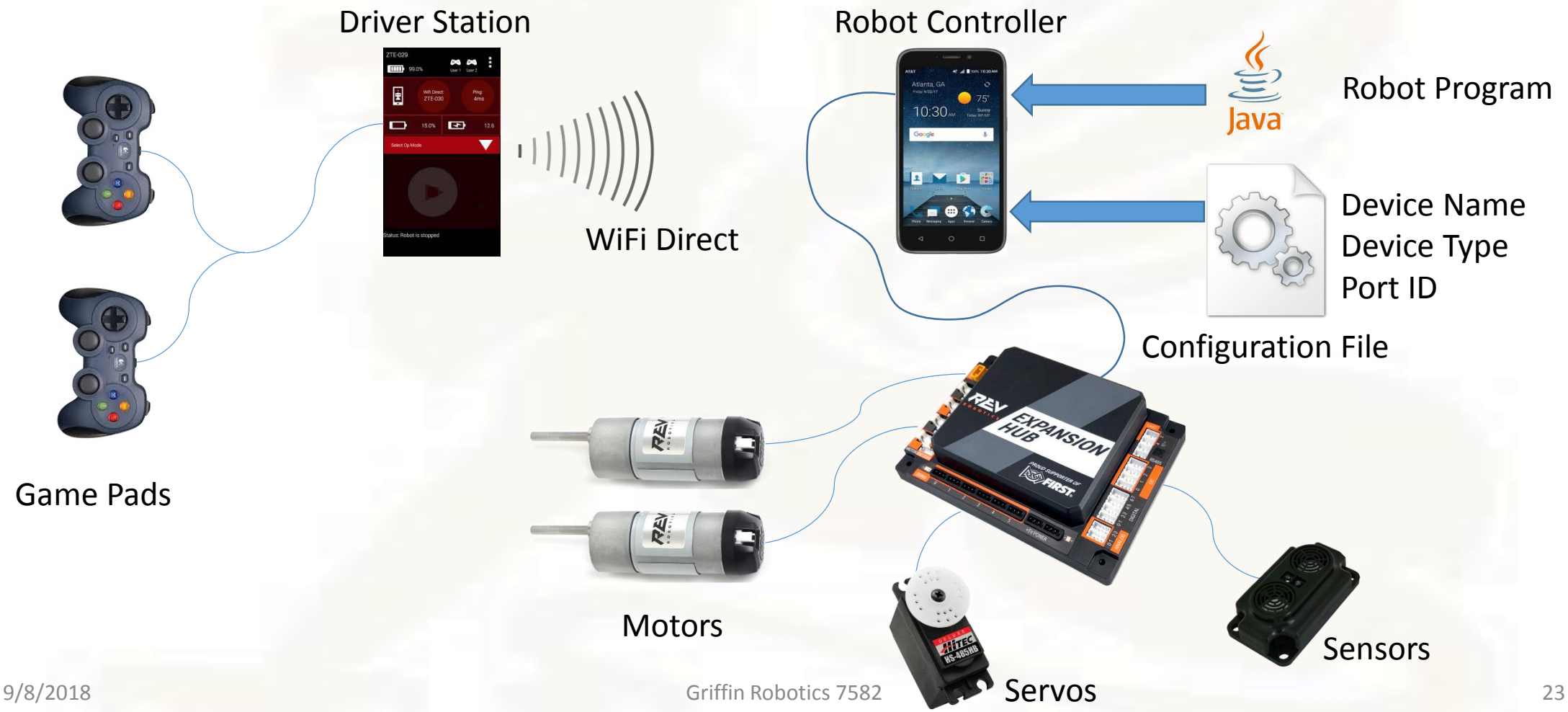
Telemetry

- Telemetry sends data from the robot to the driver station for display
- The telemetry object is inherited from OpMode or LinearOpMode
- `telemetry.addData(String caption, String value)`
- `telemetry.addData(String caption, String format, args…)`
- `telemetry.update()`
  - In Iterative Op Mode, update occurs automatically at the end of each loop

Agenda

1. 1:00 – 1:55: Robot Programming Basics
   - Linear (Autonomous) Programming Model
   - Iterative (Driver) Programming Model
   - Talking To The Robot
2. 2:00 – 2:55: Advanced Programming
   - Using Inheritance And Abstraction To Support Multiple Robots
   - Using Exceptions To Handle Hardware Failures
3. 3:00 – 3:55: Special Topics
   - Open Discussion And Q/A

Robot Architecture



Initializing Robot Devices In Java

- All devices (motors, servos, sensors) are represented as Java objects
  - Motors are represented as objects of type DcMotor
  - Servos are represented as objects of type Servo
  - Continuous rotation servos are objects of type CRServo
  - REV color sensors are objects of type ColorSensor
  - MR ultrasonic range sensors are objects of type DistanceSensor
- Import device types from com.qualcomm.robotcore.hardware.*
- Declare devices as class objects with initial values of *null*
  - `DcMotor dcm = null;`
- Initialize devices in `init()` or initialization section of the robot program
- Use `hardwareMap.get()` to initialize the objects
  - `hardwareMap` is a class object of the parent class `OpMode` or `LinearOpMode`
  - `DcMotor dcm = hardwareMap.get(DcMotor.class, "xyz");`
  - Where "`xyz`" is the name of the device as it is used in the configuration file
  - `DcMotor.class` is the Java type of the device object

Configuration Files

- Bridge the gap between devices in your program and actual hardware
- Associate a device name (used by the program) with a device type and a port identifier (used by the hardware)
- Each time you call `hardwareMap.get()`:
  - The robot program asks the FTC software for a device by its name and type
  - The FTC software checks the name and type against the configuration file
  - If all is well, it returns a handle that can talk to the hardware on its port
  - OR, it throws an Exception when the name is missing, the type doesn't match, or there is some other problem talking to the hardware device

DcMotors – Run Using Encoder

```
DcMotor drive = null;                    // class declaration

                                         // initialization
drive = hardwareMap.get(DcMotor.class, "drive name");
drive.setDirection(DcMotor.Direction.FORWARD);
drive.setPower(0);
drive.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
drive.setMode(DcMotor.RunMode.RUN_USING_ENCODER);

drive.getCurrentPosition();              // use, read the encoder value
drive.setPower(DRIVE_POWER);             // -1.0 < power < 1.0
```

DcMotors – Run To Position

```
DcMotor lift = null;                    // class declaration

                                        // initialization
lift = hardwareMap.get(DcMotor.class, "lift name");
lift.setDirection(DcMotor.Direction.FORWARD);
lift.setPower(0);
lift.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
lift.setMode(DcMotor.RunMode.RUN_TO_POSITION);

lift.setTargetPosition(target);         // use, target = encoder value
lift.setPower(LIFT_POWER);              // -1.0 < power < 1.0
```

Servos

```
Servo tail = null;                          // class declaration

                                            // initialization
tail = hardwareMap.get(Servo.class, “servo name”);
tail.setDirection(Servo.Direction.FORWARD);

tail.setPosition(TAIL_POSITION);     // use, -1.0 < position < 1.0
```

REV Color/Range Sensor

```
ColorSensor    color = null;          // class declaration
DistanceSensor range = null;

                                      // initialization
color = hardwareMap.get(ColorSensor.class,    “c/r sensor name”);
range = hardwareMap.get(DistanceSensor.class, “c/r sensor name”);

double alpha = color.alpha();         // use
double red   = color.red();
double green = color.green();
double blue  = color.blue();
double range = range.getDistance(DistanceUnit.INCH);
```

Modern Robotics Ultrasonic Range Sensor

```
DistanceSensor sensor = null;          // class declaration

                                       // initialization
sensor = hardwareMap.get(DistanceSensor.class, “sensor name”);

                                       // use
double range = sensor.getDistance(DistanceUnit.INCH);
```

Fields In gamepad1 & gamepad2

Field	Type	Field	Type	Field	Type
a	boolean	left_bumper	boolean	left_stick_x	double
b	boolean	right_bumper	boolean	left_stick_y	double
x	boolean	left_stick_button	boolean	right_stick_x	double
y	boolean	right_stick_button	boolean	right_stick_y	double
dpad_up	boolean	start	boolean	left_trigger	double
dpad_down	boolean	back	boolean	right_trigger	double
dpad_left	boolean	guide	boolean		
dpad_right	boolean				

To use, say: `gamepad`*`k`*`.`*`field`*, for example, `gamepad1.right_bumper`

Analog Sensor (E.g., Pololu IR Sensor)

```
AnalogInput sensor = null;                  // class declaration

                                            // initialization
sensor = hardwareMap.get(AnalogInput.class, “sensor name”);

                                            // use
double voltage = sensor.getVoltage();
                                            // you must then convert the
                                            // voltage to a distance or
                                            // other appropriate value
```

Agenda

1. 1:00 – 1:55: Robot Programming Basics
   - Linear (Autonomous) Programming Model
   - Iterative (Driver) Programming Model
   - Talking To The Robot
2. 2:00 – 2:55: Advanced Programming
   - Using Inheritance And Abstraction To Support Multiple Robots
   - Using Exceptions To Handle Hardware Failures
3. 3:00 – 3:55: Special Topics
   - Open Discussion And Q/A

Why use abstraction and inheritance?

- Your team has two different ideas you want to try, like two types of drives
- You want to try both out and compare them side by side
- You copy your Op Mode and make changes to the hardware
- Next you find a bug and need to update the controls of both robots…
- With every change you make, it gets harder to keep them the same
- *Or*…
- Your Op Mode is nearly done when you find some better hardware
- To change out the old hardware you need to find every place that touches the old and replace it with new controls. It is scattered everywhere!
- There must be a better way! (There is.)

What is inheritance?

- Java supports an idea called *inheritance*
- Two classes may have a parent-child relationship
- A *child class* (or *subclass*) *inherits* all of the data and subroutines from its *parent class* (or *superclass*)
- Examples you have already seen
  - Your Driver Op Mode inherits from class OpMode
  - Your Autonomous Op Mode inherits from class LinearOpMode
  - In other words, your Driver Op Mode is a subclass of superclass OpMode and your Autonomous Op Mode is a subclass of superclass LinearOpMode
- Everything in the superclass (parent) is also in the subclass (child)

Inheritance Example

```
public class Parent {                    // parent class (or superclass)
    public void move(double bearing, double speed) {
        …
    }
}

public class Child extends Parent {    // child class (or subclass)
    // Child can use move from Parent and spin from Child
    public void spin(double speed) {
        …
    }
}
```

Override Example

- Functions in the child class can override functions in the parent, too

```
public class Parent {                    // parent class (or superclass)
    public void move(double bearing, double speed) {
        …
    }
}

public class Child extends Parent {    // child class (or subclass)
    // Child can use move from Child or super.move from Parent
    @Override
    public void move(double bearing, double speed) {
        …
    }
}
```

Abstract Functions And Classes

- Functions can be declared abstract in a parent class
- Abstract functions are declared but not implemented , e.g.,

```
public abstract void move(double bearing, double speed);
```

- Declaring a function abstract says the **child** must implement it
- Any class that contains an abstract function must be declared abstract
- Any child of an abstract class must implement the abstract function

Abstract Class Example

```
public abstract class AbstractParent {          // abstract parent class
    public abstract void move(double bearing, double speed);
}

public class Child extends AbstractParent {     // child class
    @Override
    public void move(double bearing, double speed) {
        …
    }
}
```

Objects Of Abstract Type

- Variables of abstract classes can only be assigned child classes

```
AbstractParent data = new Child();
```

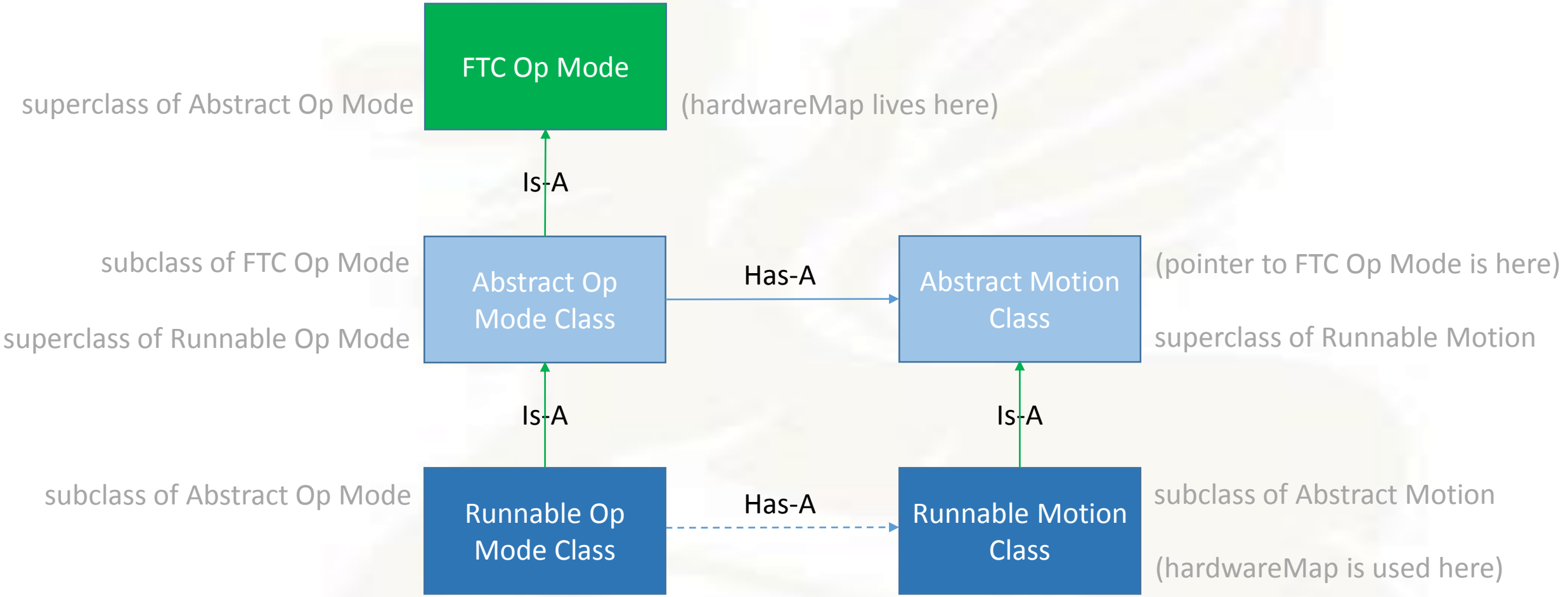
- Variables of abstract classes may use only routines and data from the abstract class (or its parents)

This sounds complicated, how do I use it?

Instead of putting everything together in one Op Mode class…

1. Create an abstract motion class that defines *what* your drive motors do
2. Create a non-abstract child of the abstract motion class (from 1) that
    1. Implements the abstract functions of the parent class, and
    2. Talks to the real hardware
3. Create an abstract parent Op Mode that
    1. Takes input from the gamepads, sensors, etc., and
    2. Uses the abstract motion class (step 1) to define when and how the robot moves
4. Create a child class of the abstract Op Mode that
    1. Sets the abstract motor object to the motor class (from step 2)

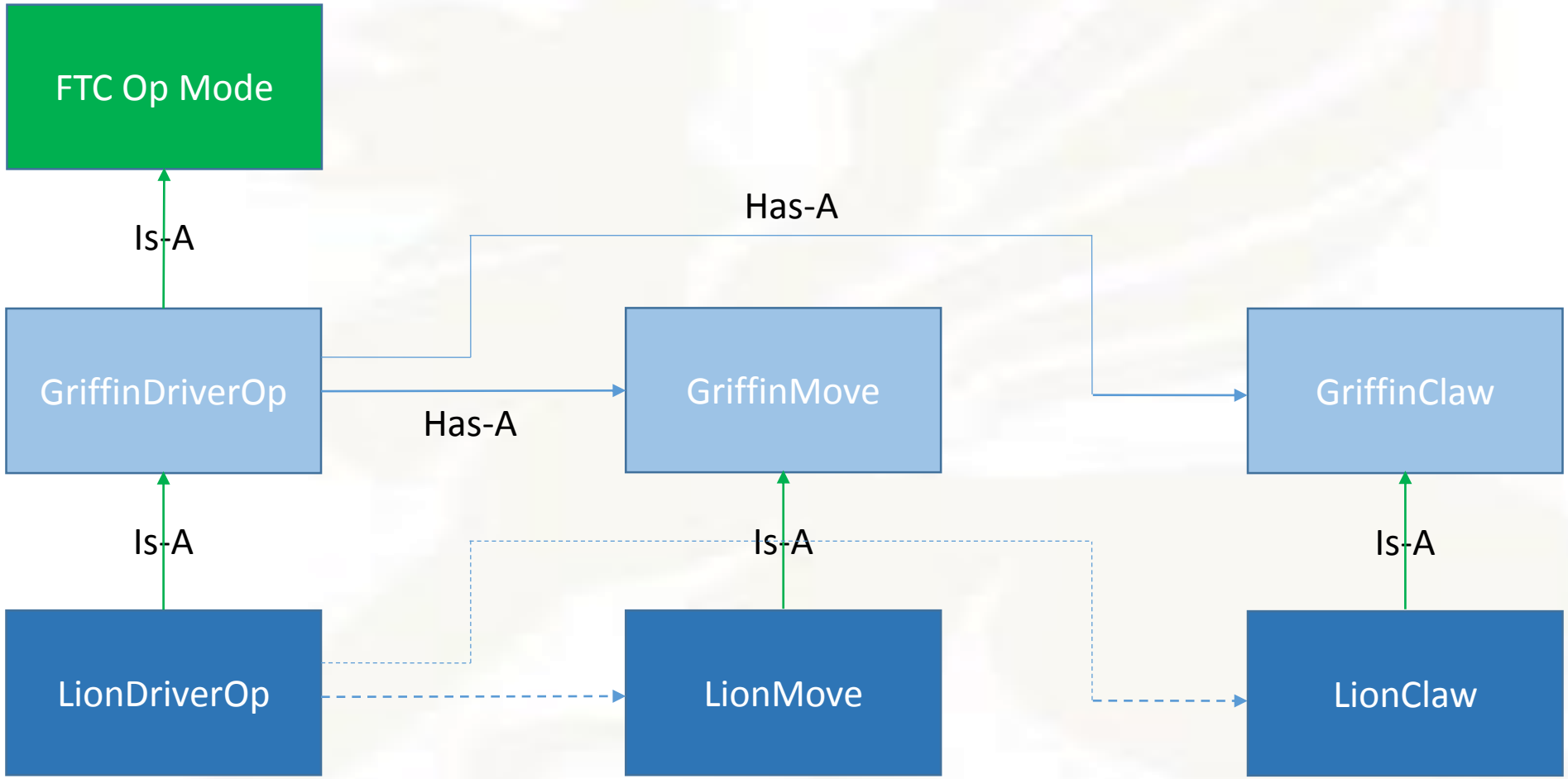
A Better Robot Design



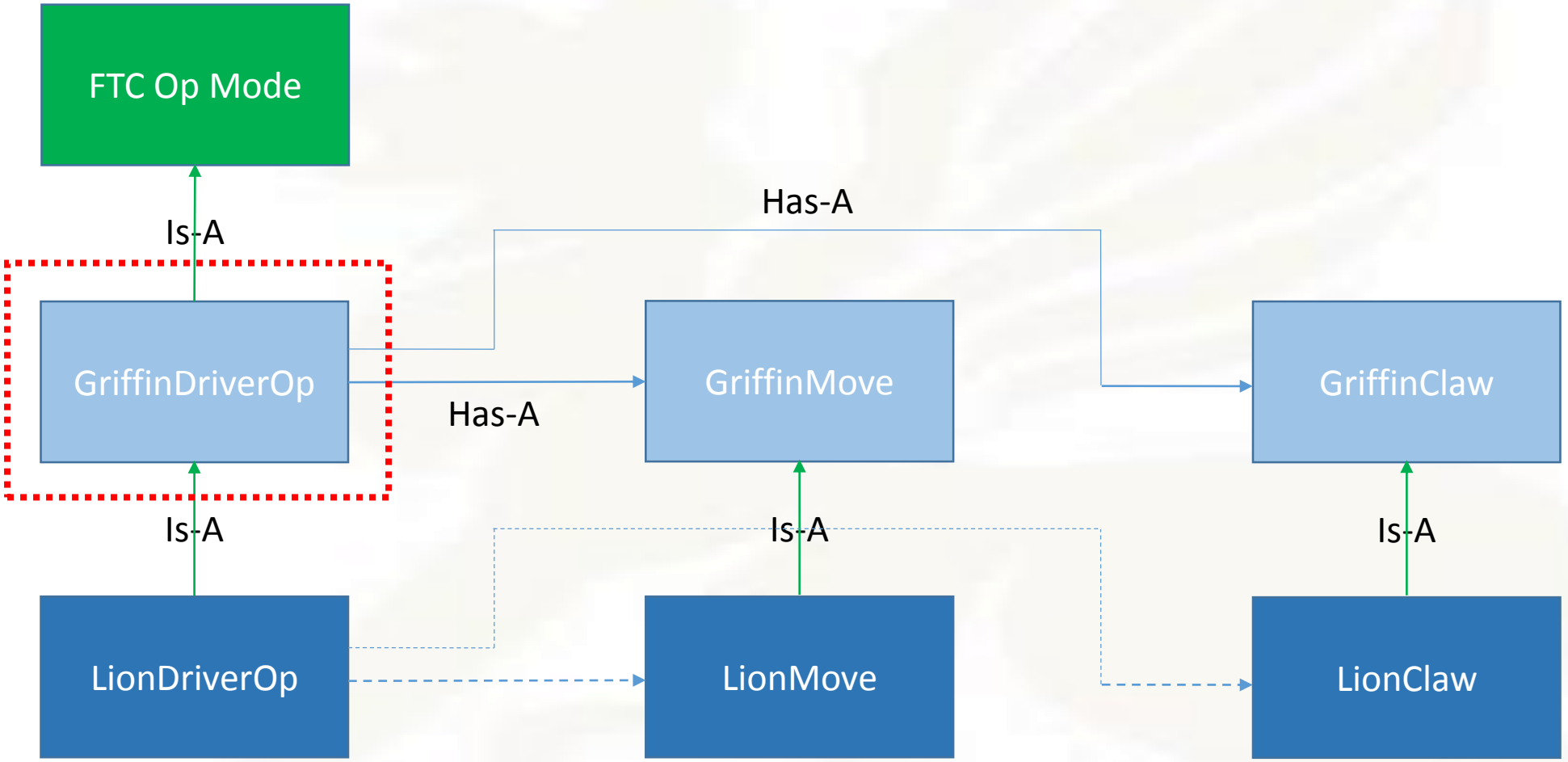
It sounds like more work, how is this easier?

- You write your Op Mode in terms of what the motors do for you (e.g., move forward, etc.) instead of how you have to talk to the hardware.
  - Defining your robot program in terms of *what it does* instead of *how it works* makes it easier to understand and simpler to code.
- To support two different types of robot, re-write only those parts that are different, e.g., write a new runnable motion class (step 2) and create a new runnable Op Mode class (step 4).
  - The abstract Op Mode and abstract motion class are shared.
  - This is much easier than re-writing the Op Mode each time the hardware changes.
- Replacing old hardware with more advanced (e.g., tank drive with holonomic drive) is easier because you change it in only one place.
  - Change the runnable motion class from step 2, the rest stays the same.

Example – Griffin Robot With Claw



Griffin Robot With Claw - GriffinDriverOp



Griffin Driver Op Mode

```
public class GriffinDriverOp extends OpMode {
    public  GriffinMove drive   = null;
    public  GriffinClaw claw    = null;
    private ElapsedTime runtime = new ElapsedTime();

    @Override
    public void init() {
        . . .
    }

    @Override
    public void start() { runtime.reset(); }


    @Override
    public void loop() {
        . . .
    }

    @Override
    public void stop() {
        . . .
    }
}
```

Griffin Driver Op Mode - Init

```
. . .

@Override
public void init() {
    telemetry.addData("Status", "Initializing Drive.");
    drive.init();

    telemetry.addData("Status", "Initializing Claw.");
    claw.init();

    telemetry.addData("Status", "Initialization Complete.");
}

. . .
```

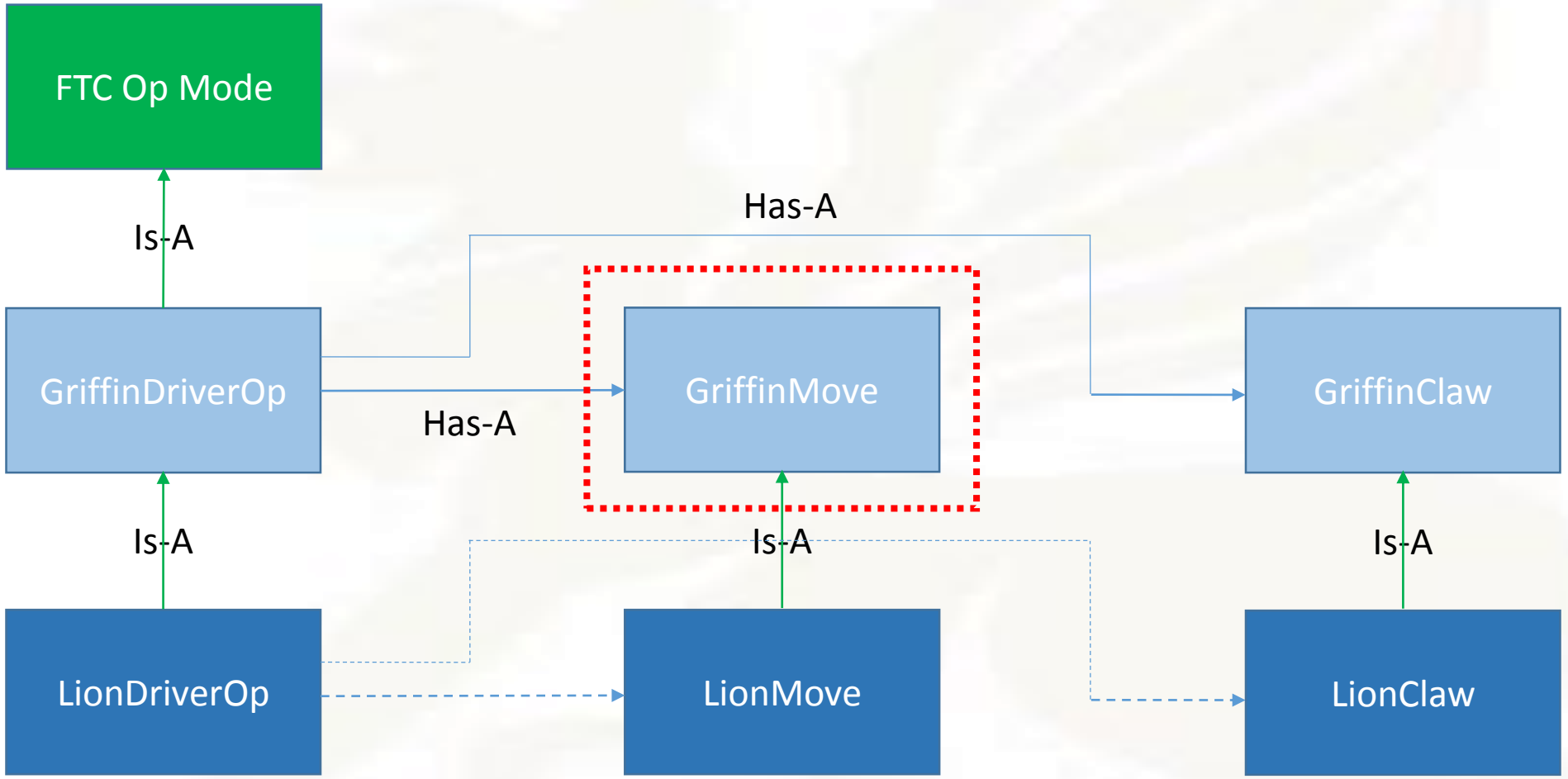
Griffin Driver Op Mode - Loop

```
. . .
@Override
public void loop() {
    if (gamepad1.dpad_up) {
        drive.move(  0.0, 0.5);
    } else if (gamepad1.dpad_right) {
        drive.move( 90.0, 0.5);
    } else if (gamepad1.dpad_down) {
        drive.move(180.0, 0.5);
    } else if (gamepad1.dpad_left) {
        drive.move(270.0, 0.5);
    } else {
        drive.move(  0.0, 0.0);
    }

    if (gamepad1.x) {
        claw.open();
    } else if (gamepad1.b) {
        claw.close();
    }
}
. . .
```

Griffin Driver Op Mode - Stop

```
    . . .
    @Override
    public void stop() {
        telemetry.addData("Status", "Stopping Drive.");
        drive.stop();

        telemetry.addData("Status", "Stopping Claw.");
        claw.stop();

        telemetry.addData("Status", "Robot Stopped.");
    }
}
```

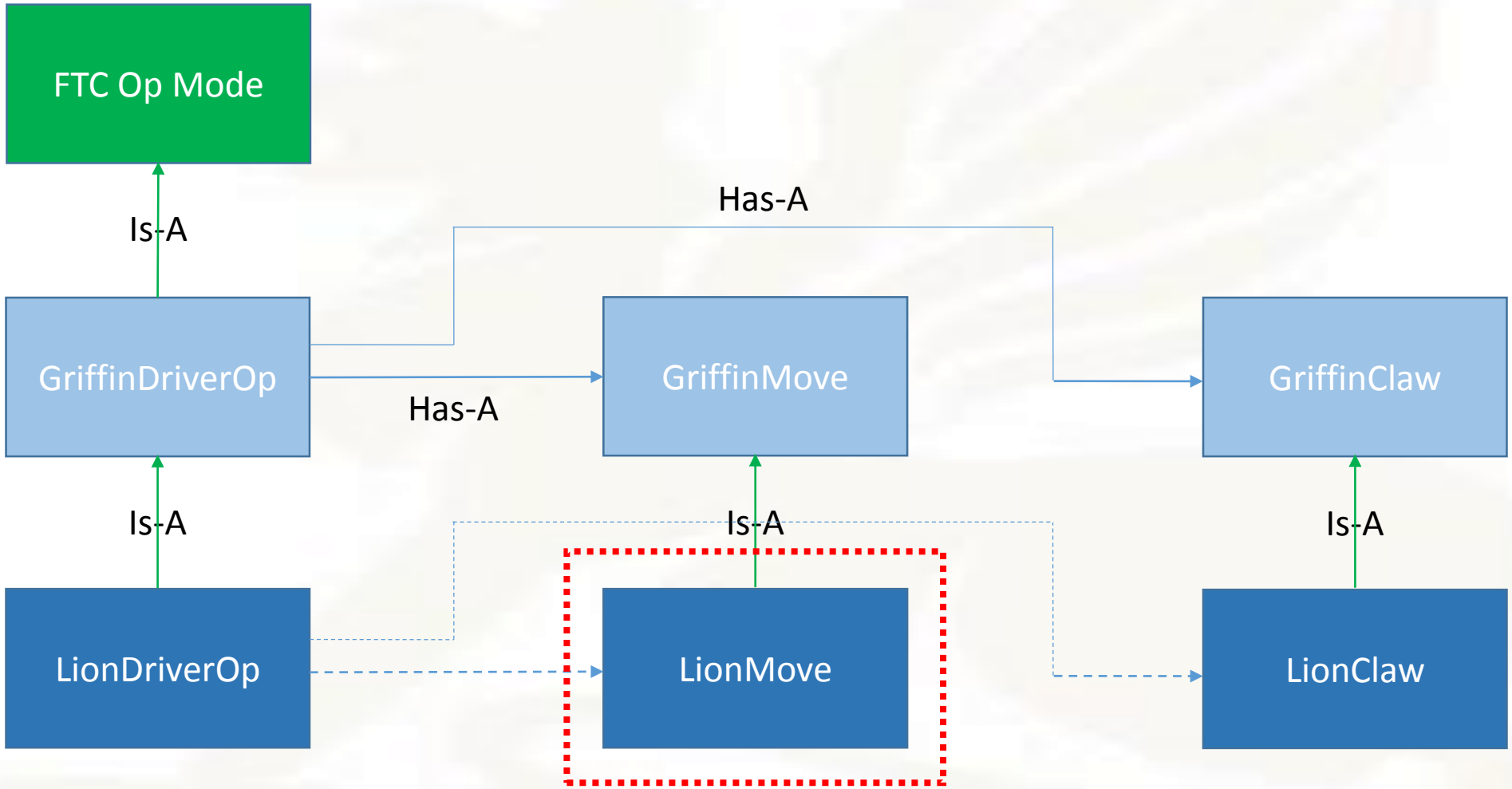
Griffin Robot With Claw - GriffinMove



Griffin Move

```
public abstract class GriffinMove {
    protected OpMode op_mode = null;

    public abstract void init();
    public abstract void move(double bearing, double speed);
    public abstract void stop();

    public GriffinMove(OpMode op) {
        op_mode = op;      // save a pointer to the FTC Op Mode
    }
}
```

Griffin Robot With Claw - LionMove



Lion Move

```
public class LionMove extends GriffinMove {

    public LionMove(OpMode op) {
        super(op);
    }

    private DcMotor front_left  = null;
    private DcMotor front_right = null;
    private DcMotor back_right  = null;
    private DcMotor back_left   = null;

    public void init() {
        . . .
    }

    public void move(double bearing, double power) {
        . . .
    }

    public void stop() {
        . . .
    }
}
```

Lion Move – Init

```
public void init() {
    front_left = op_mode.hardwareMap.get(DcMotor.class, "front left");
    front_left.setDirection(DcMotor.Direction.FORWARD);
    front_left.setPower(0);
    front_left.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    front_left.setMode(DcMotor.RunMode.RUN_USING_ENCODER);

    front_right = op_mode.hardwareMap.get(DcMotor.class, "front right");
    front_right.setDirection(DcMotor.Direction.FORWARD);
    front_right.setPower(0);
    front_right.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    front_right.setMode(DcMotor.RunMode.RUN_USING_ENCODER);

    back_right = op_mode.hardwareMap.get(DcMotor.class, "back right");
    back_right.setDirection(DcMotor.Direction.FORWARD);
    back_right.setPower(0);
    back_right.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    back_right.setMode(DcMotor.RunMode.RUN_USING_ENCODER);

    back_left = op_mode.hardwareMap.get(DcMotor.class, "back left");
    back_left.setDirection(DcMotor.Direction.FORWARD);
    back_left.setPower(0);
    back_left.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    back_left.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
}
```

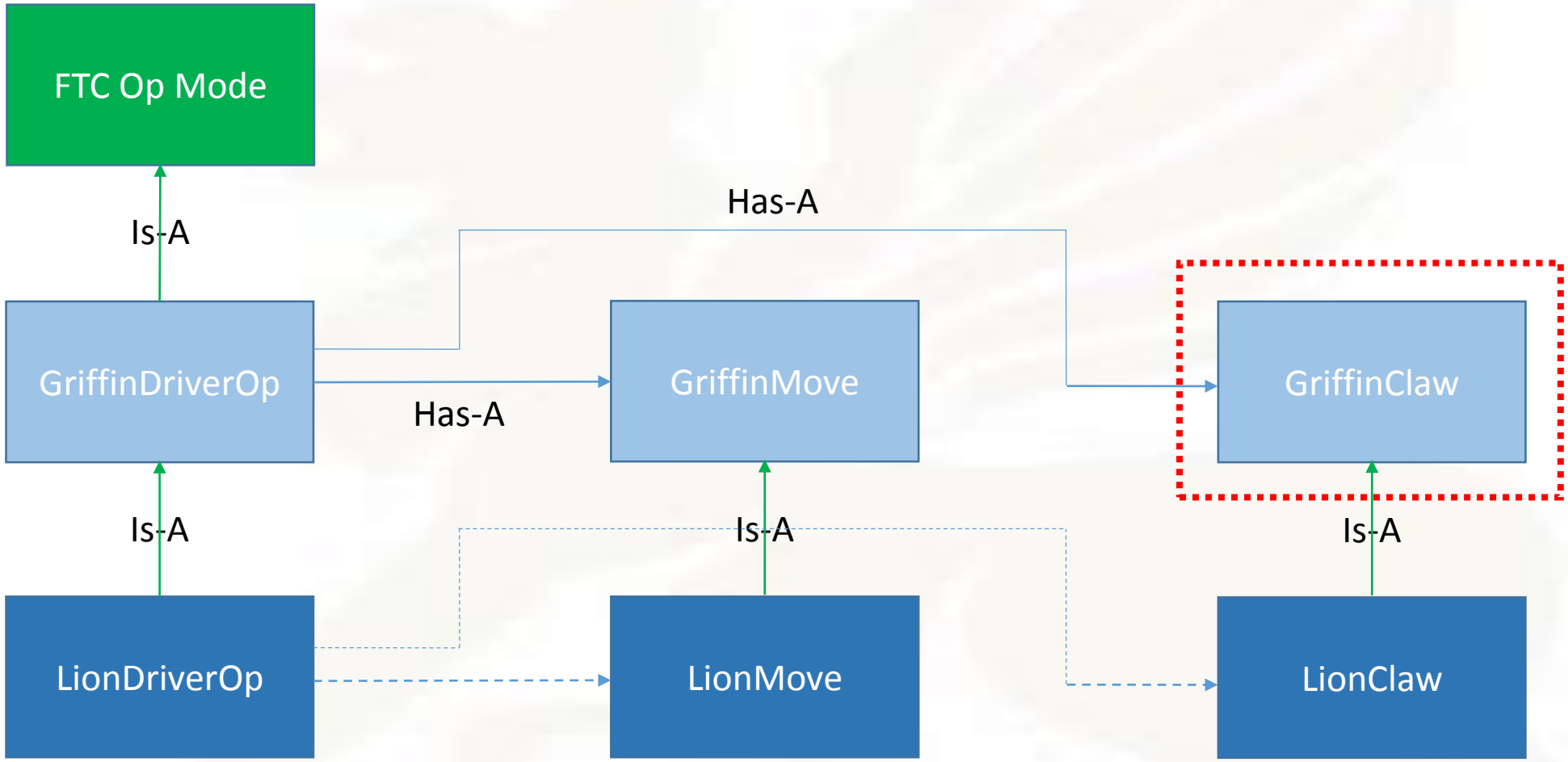
Lion Move – Move

```
public void move(double bearing, double power) {
    if (315 < bearing || bearing < 45) {           // go forward
        front_left.setPower (-power);
        front_right.setPower( power);
        back_right.setPower ( power);
        back_left.setPower  (-power);
    } else if (45 < bearing && bearing < 135) {    // go right
        front_left.setPower (-power);
        front_right.setPower(-power);
        back_right.setPower ( power);
        back_left.setPower  ( power);
    } else if (135 < bearing && bearing < 225) {   // go backward
        front_left.setPower ( power);
        front_right.setPower(-power);
        back_right.setPower (-power);
        back_left.setPower  ( power);
    } else if (225 < bearing && bearing < 315) {   // go left
        front_left.setPower ( power);
        front_right.setPower( power);
        back_right.setPower (-power);
        back_left.setPower  (-power);
    }
}
```

Lion Move – Stop

```
public void stop() {
    front_left.setPower (0);
    front_right.setPower(0);
    back_right.setPower (0);
    back_left.setPower  (0);
}
```

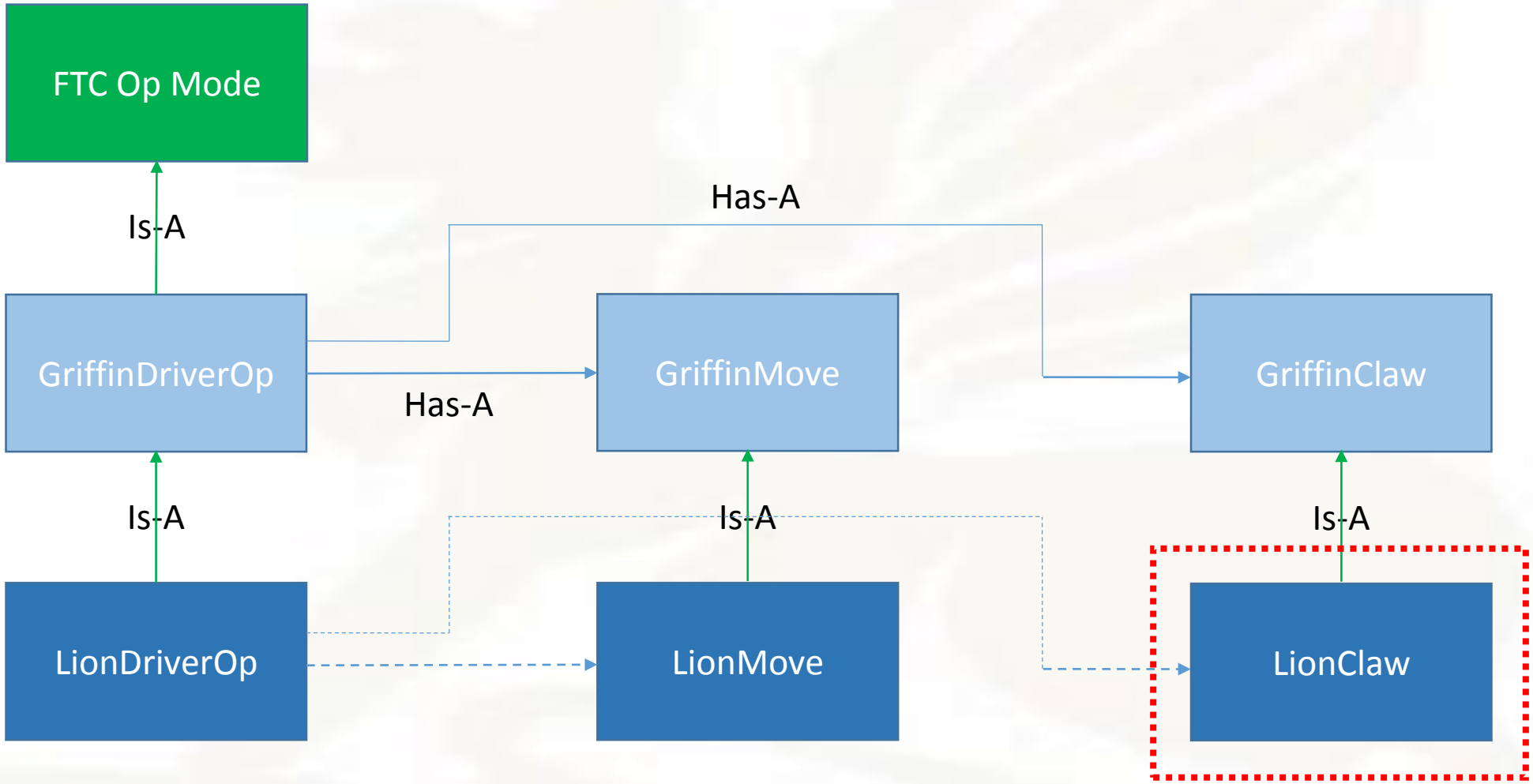
Griffin Robot With Claw - GriffinClaw



Griffin Claw

```
public abstract class GriffinClaw {
    public OpMode op_mode = null;

    public abstract void init();
    public abstract void open();
    public abstract void close();
    public abstract void stop();

    public GriffinClaw(OpMode op) {
        op_mode = op;       // save a pointer to the FTC Op Mode
    }
}
```

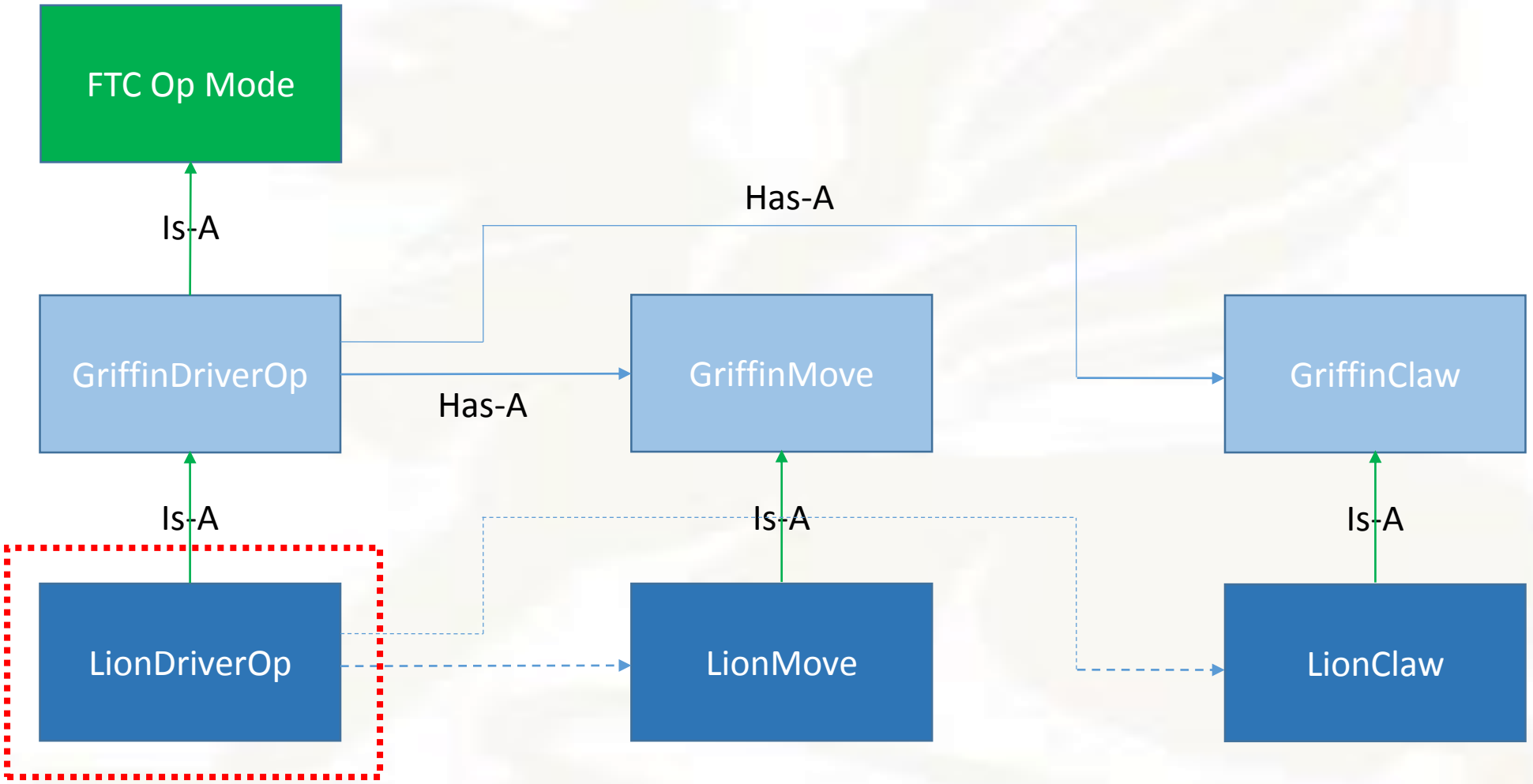
Griffin Robot With Claw - LionClaw



Lion Claw

```
public class LionClaw extends GriffinClaw {

    public LionClaw(OpMode op) {
        super(op);
    }

    private Servo claw = null;

    public void init() {
        claw = hardwareMap.get(Servo.class, "claw");
        claw.setDirection(Servo.Direction.FORWARD);
    }

    public void open() {
        claw.setPosition(1.0);
    }


    public void close() {
        claw.setPosition(0.0);
    }


    public void stop() {
        close();
    }
}
```

Griffin Robot With Claw - LionDriverOp



Lion Driver Op

```
@TeleOp(name="Lion Driver Op Mode", group="Iterative Opmode")
public class LionDriverOp extends GriffinDriverOp {
    public LionDriverOp() {
        drive = new LionMove(this);
        claw  = new LionClaw(this);
    }
}
```

Agenda

1. 1:00 – 1:55: Robot Programming Basics
   - Linear (Autonomous) Programming Model
   - Iterative (Driver) Programming Model
   - Talking To The Robot
2. 2:00 – 2:55: Advanced Programming
   - Using Inheritance And Abstraction To Support Multiple Robots
   - Using Exceptions To Handle Hardware Failures
3. 3:00 – 3:55: Special Topics
   - Open Discussion And Q/A

What could possibly go wrong?

1. You forget to initialize a device
2. You use the wrong name for a device
3. You use the wrong device type (e.g., you call a sensor a servo)
4. The device isn’t plugged in
5. The device breaks

What happens when something goes wrong?

- In every case your program stops what it's doing and "throws an exception"
- If you do nothing, your program will abort with a nasty error message
- If you catch the exception, you can deal with it and continue
- The way we catch exceptions is with a try/catch block

Catching An Exception

```
try {
    drive = hardwareMap.get(DcMotor.class, “drive name”);
    drive.setDirection(DcMotor.Direction.FORWARD);
    drive.setPower(0);
    drive.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    drive.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
} catch (Exception e) {
    drive = null;
}
. . .
if (drive != null) {
    drive.setPower(power);
}
```

Agenda

1. 1:00 – 1:55: Robot Programming Basics
   - Linear (Autonomous) Programming Model
   - Iterative (Driver) Programming Model
   - Talking To The Robot
2. 2:00 – 2:55: Advanced Programming
   - Using Inheritance And Abstraction To Support Multiple Robots
   - Using Exceptions To Handle Hardware Failures
3. 3:00 – 3:55: Special Topics
   - Open Discussion And Q/A

Open Discussion and Q/A