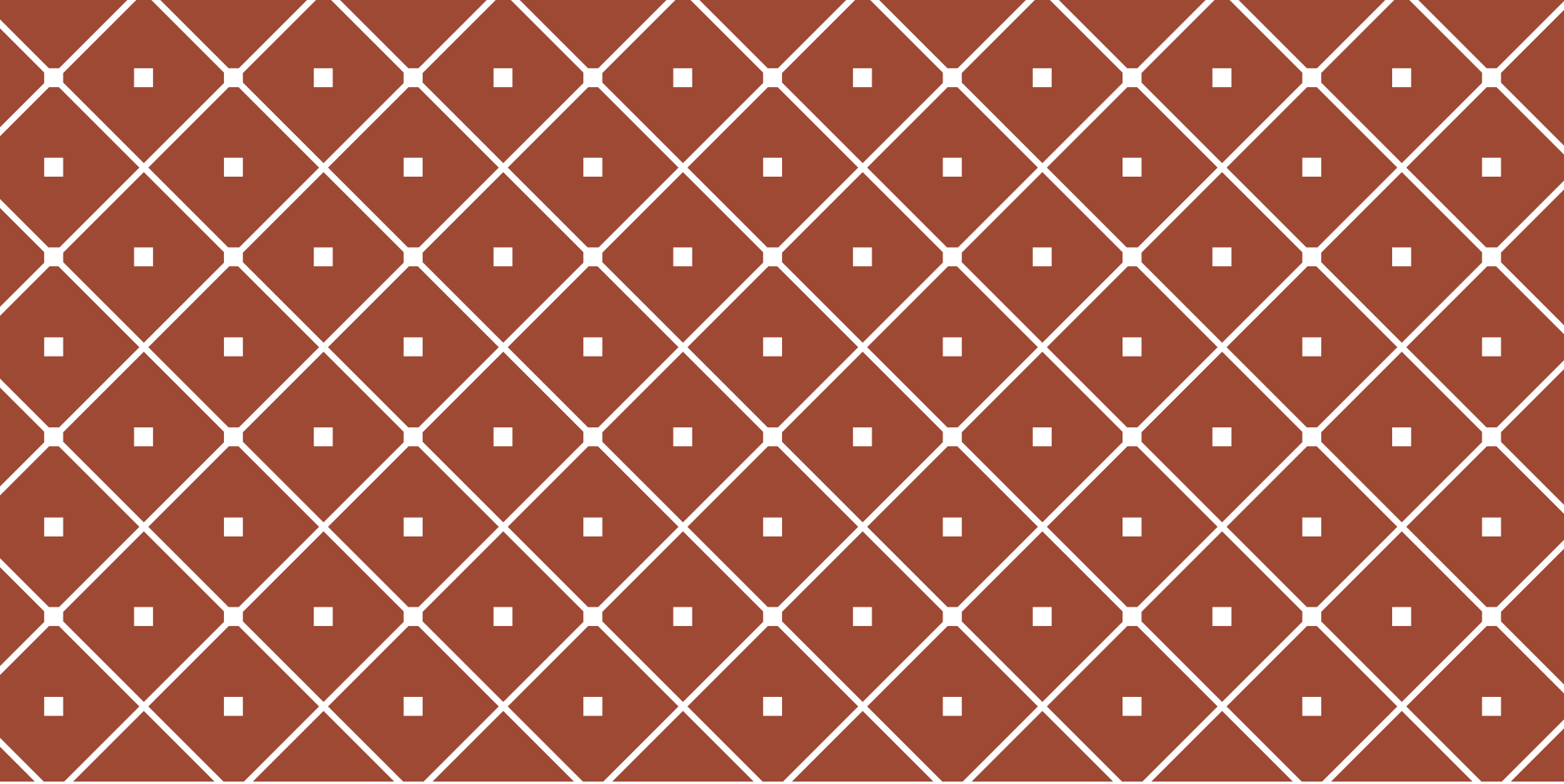


# CONTROLS-PROGRAMMING TRAINING

2019 Season

# WHAT TO KEEP IN MIND

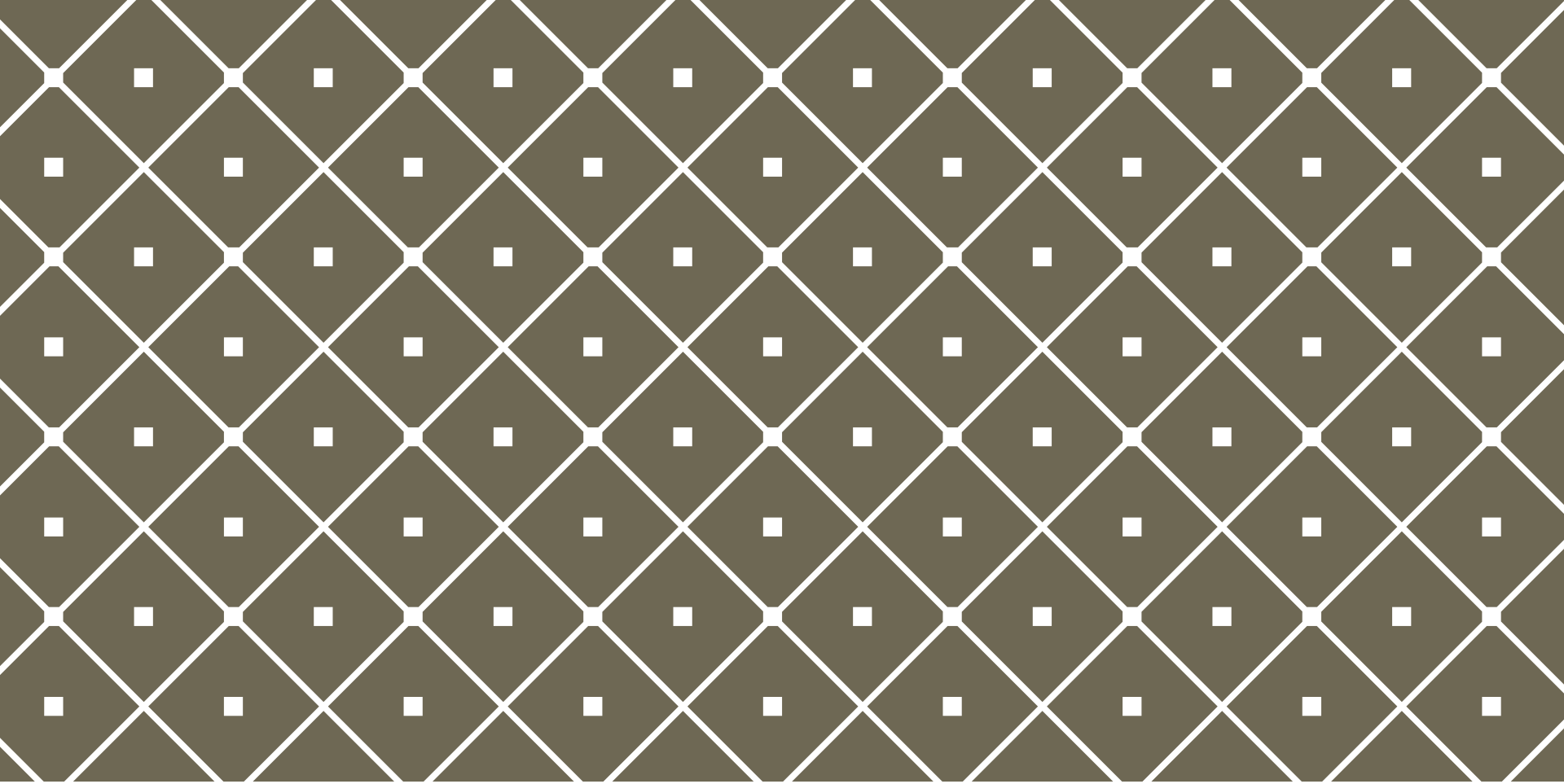
Two Main Game Segments

- **Autonomous:** 15 seconds
  - No human input allowed
  - Robot must make all decisions
- **Teleoperated:** 2 minutes 15 seconds
  - Human driver controls robot
  - Robot can assist driver with camera or sensor readings

Programming must work closely with drive team

- They have final say over **controls** and **driving mode**
- They tell us how they want the robot to work

The robot is programmed with **Java**



# FRC SPECIFIC CLASSES/METHODS

# REMEMBER…

The class names are just referring to the generic class that is going to be **inherited** by our program. The actual code we write will be contained in our **subclass** that inherits its **methods** from the generic class.

# ROBOT.JAVA

## Class Usage

This is the main class

It can be considered the starting point

Every other class is created by Robot.java

Each robot program **must** have a Robot.java implementation

## Important Methods

`robotInit()`

- Called when robot is first turned on

`teleopInit()/autoInit()`

- Called when teleop and auto are started

`teleopPeriodic/autoPeriodic()`

- Repeatedly called during teleop and auto
- Must include `Scheduler.getInstance().run()`

```
Subsystem sub = new Subsystem();
```

- All subsystems should be initialized in Robot.java

# EXAMPLES

```
public static OI oi;
public static Pneumatics pneumatics;
public static DriveTrain driveTrain;
public static Elevator elevator;
public static AHRS gyro;
public static frc.team1329.sensors.Ultrasonic ultrasonic;
public static PushButton pushButton;
private Command autoCommand;

private static DigitalInput autoSelectSwitchOne;
private static DigitalInput autoSelectSwitchTwo;

public static boolean advisableToClimb = true;

@Override
public void robotInit() {
    oi = new OI();
    driveTrain = new DriveTrain();
    elevator = new Elevator();
    gyro = new AHRS(SPI.Port.kMXP);
    pneumatics = new Pneumatics();
    ultrasonic = new Ultrasonic();

    autoSelectSwitchOne = new DigitalInput(RobotMap.AUTO_SWITCH_ONE);
    autoSelectSwitchTwo = new DigitalInput(RobotMap.AUTO_SWITCH_TWO);
      pushButton = new PushButton(0);
}
```

```
@Override
public void autonomousInit() {
    Robot.gyro.reset();
    String message = DriverStation.getInstance().getGameSpecificMessage();
    this.autoCommand = setAutoCommand(message.charAt(0), message.charAt(1));
    this.autoCommand.start();
      (new AutoDriveGroupExample()).start();
      (new AutoDriveStraight(5.0)).start();
      (new AutoDriveDistance(10.0)).start();
      (new AutoGroup()).start();
}
```

```
@Override
public void autonomousPeriodic() {
    Scheduler.getInstance().run();
    SmartDashboard.putNumber("DISTANCE", driveTrain.getAvgPosition());
    SmartDashboard.putNumber("GYRO ANGLE", gyro.getYaw());
}
```

```
@Override
public void teleopInit() {
    Robot.gyro.resetDisplacement();
    Robot.driveTrain.resetPositions();
}

@Override
public void teleopPeriodic() {
    Scheduler.getInstance().run();
    SmartDashboard.putNumber("FRONT LEFT SPEED",driveTrain.getSpeeds(true)[FRONT_LEFT]);
    SmartDashboard.putNumber("FRONT RIGHT SPEED",driveTrain.getSpeeds(true)[FRONT_RIGHT]);
    SmartDashboard.putNumber("BACK LEFT SPEED",driveTrain.getSpeeds(true)[BACK_LEFT]);
    SmartDashboard.putNumber("BACK RIGHT SPEED",driveTrain.getSpeeds(true)[BACK_RIGHT]);

    SmartDashboard.putNumber("FRONT LEFT POSITION",driveTrain.getPositions()[FRONT_LEFT]);
    SmartDashboard.putNumber("FRONT RIGHT POSITION",driveTrain.getPositions()[FRONT_RIGHT]);
    SmartDashboard.putNumber("BACK LEFT POSITION",driveTrain.getPositions()[BACK_LEFT]);
    SmartDashboard.putNumber("BACK RIGHT POSITION",driveTrain.getPositions()[BACK_RIGHT]);
```

# SUBSYSTEM.JAVA

## Class Usage

Subsystems are a virtual representation of a physical group of hardware on the robot

They contain multiple objects that represent something like a drivetrain, a claw, or any other physical system

They are how the program can access and control the actual robot

## Important Methods

Initializer; `public Subsystem()`

- Should include initialization of all the hardware objects held by the subsystem

`initDefaultCommand()`

- Binds a specific command to the subsystem

All other methods are subsystem specific, they are custom written based on what tasks the subsystem may perform

# EXAMPLES

```
public DriveTrain(){
    super("DriveTrain");
    frontLeft = new WPI_TalonSRX(DRIVE_MOTOR_IDS[FRONT_LEFT]);
    frontLeftFollower = new WPI_TalonSRX(FOLLOWER_MOTOR_IDS[FRONT_LEFT]);
    frontLeftFollower.setInverted(true);
    frontLeftFollower.follow(initTalon(frontLeft,true));

    frontRight = new WPI_TalonSRX(DRIVE_MOTOR_IDS[FRONT_RIGHT]);
    frontRightFollower = new WPI_TalonSRX(FOLLOWER_MOTOR_IDS[FRONT_RIGHT]);
    frontRightFollower.follow(initTalon(frontRight, false));

    backLeft = new WPI_TalonSRX(DRIVE_MOTOR_IDS[BACK_LEFT]);
    backLeft.setSensorPhase(true);
    backLeftFollower = new WPI_TalonSRX(FOLLOWER_MOTOR_IDS[BACK_LEFT]);
    backLeftFollower.setInverted(true);
    backLeftFollower.follow(initTalon(backLeft, true));
```

```
    @Override
    protected void initDefaultCommand() {
        setDefaultCommand(new DriveWithGamepad());
          setDefaultCommand(new DriveWithFlightController());
    }
```

```
public void mecDrive(double x, double y, double rot, double gyro, boolean pid){
    //        if(fieldOriented) {
          pX = (x * Math.cos(Math.toRadians(gyro))) + (y * Math.sin(Math.toRadians(gyro)));
          pY = -(x * Math.sin(Math.toRadians(gyro))) + (y * Math.cos(Math.toRadians(gyro)));
      }                                 //
    ControlMode controlMode = pid ? ControlMode.Velocity : ControlMode.PercentOutput;
    double multiplier = pid ? (MAX_SPEED/TALON_POSITION_MULTIPLIER)/10.0 : 1.0;
    frontLeft.set(controlMode, frontLeftRamper.scale(normalize(x + y + rot)) * multiplier);
    frontRight.set(controlMode, frontRightRamper.scale(normalize(x + y - rot)) * multiplier);
    backLeft.set(controlMode, backLeftRamper.scale(normalize(-x + y + rot)) * multiplier);
    backRight.set(controlMode, backRightRamper.scale(normalize(-x + y - rot)) * multiplier);
}
public void tankDrive(double left, double right){
    frontRight.set(ControlMode.Velocity,(right/TALON_POSITION_MULTIPLIER)/10.0);
    frontLeft.set(ControlMode.Velocity,(left/TALON_POSITION_MULTIPLIER)/10.0);
    backRight.set(ControlMode.Velocity,(right/TALON_POSITION_MULTIPLIER)/10.0);
    backLeft.set(ControlMode.Velocity,(left/TALON_POSITION_MULTIPLIER)/10.0);
}
```

# COMMAND.JAVA

## Class Usage

Commands contain any action that will be run a variable number of times on the robot

Each command contains a small bit of code to keep track of its individual action

Commands must choose a **subsystem** to use

## Important Methods

Initializer; `public Command()`

- Should contain initialization for anything used by the command
- Also includes `requires(Subsystem s)`, which tells the robot which subsystems this command uses

`execute()`

- Called repeatedly while command is running

`isFinished()`

- Checked everytime command is run, should return `true` if finished `false` if not

`end()`

- Called for cleanup after command is finished, should do stuff like stop motors that were running

# EXAMPLES

```
public DriveWithGamepad(){
    super("DriveWithGamepad");
    this.requires(Robot.driveTrain);
    this.joystick = Robot.oi.getXboxController();
    requires(Robot.driveTrain);
}
```

```
@Override
protected boolean isFinished() {
    System.out.println(distanceDriven);
    return this.distanceDriven >= goalDistance;
}
```

```
@Override
protected void execute() {
    this.x = cleanInput(this.joystick.getRawAxis(OI.LEFT_STICK_X_AXIS));
    this.y = cleanInput(joystick.getRawAxis(OI.LEFT_STICK_Y_AXIS));
    this.turn = (this.joystick.getRawAxis(OI.LEFT_TRIGGER) - this.joystick.
    this.gyro = Robot.gyro.getAngle();
    Robot.driveTrain.mecDrive(x, y,turn,gyro,false);
}
```

# OTHER IMPORTANT CLASSES

## OI.java

- OI = "Operator interface"
- Contains code which translates the controller input into commands for the robot

## RobotMap.java

- Contains constants which map hardware ids to the program

## Sensor Classes

- FRC does not provide a good sensor generic class, so we write our own for every sensor

## Trigger.java

- A generic class that can map any sort of stimulus to fire a Command

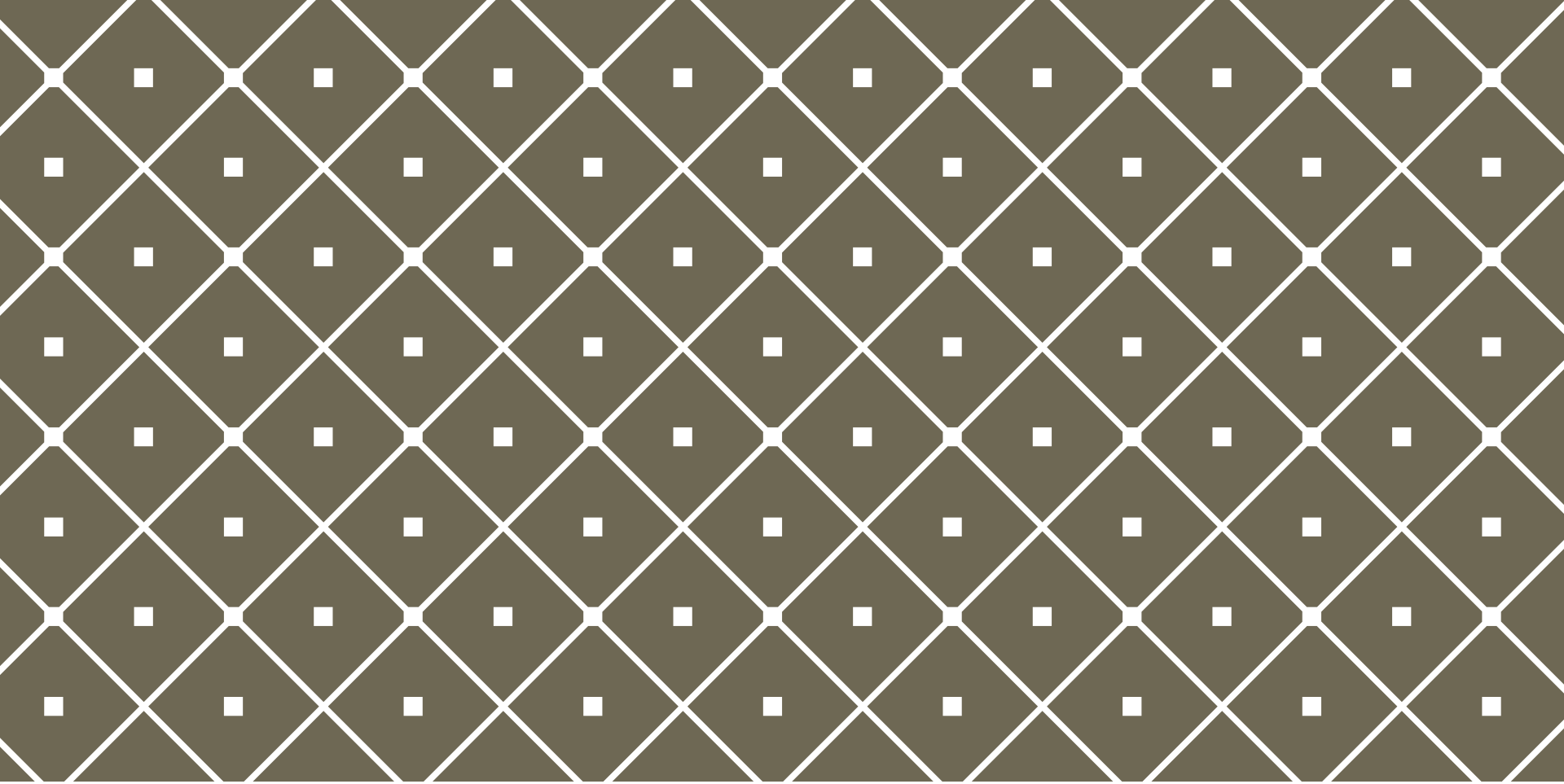
# EXAMPLES

## OI.java

```
public OI(){
    xboxController = new Joystick(2);
    flightControllerOne = new Joystick(0);
    flightControllerTwo = new Joystick(1);


    togglePneumaticButton = new JoystickButton(xboxController,A_BUTTON);
    togglePneumaticButton.whenPressed(new TogglePneumatic(1));

    fieldOrientedToggleButton = new JoystickButton(xboxController, B_BUTTON);
    fieldOrientedToggleButton.whenPressed(new ToggleFieldOrientedCommand());
}

public Joystick getXboxController() { return xboxController; }

public Joystick getFlightControllerOne() { return flightControllerOne; }

public Joystick getFlightControllerTwo() { return flightControllerTwo; }
```

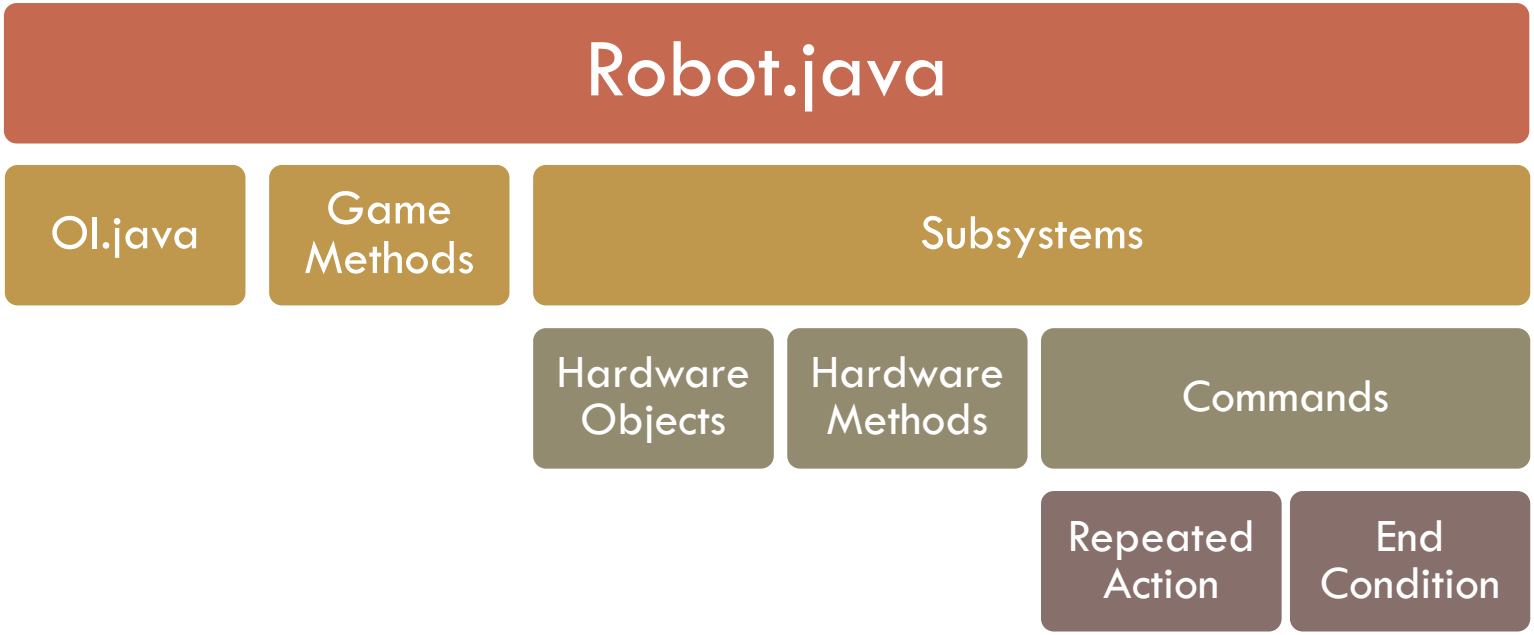
## RobotMap.java

```
public class RobotMap {
    public static final int FRONT_LEFT = 0;
    public static final int FRONT_RIGHT = 1;
    public static final int BACK_LEFT = 2;
    public static final int BACK_RIGHT = 3;
    public static final int[] DRIVE_MOTOR_IDS = {0, 2, 4, 6};
    public static final int[] FOLLOWER_MOTOR_IDS = {1, 3, 5, 7};
    public static final int ELEVATOR_VICTOR_ONE = 1;
    public static final int ELEVATOR_VICTOR_TWO = 2;
    public static final int ELEVATOR_CLAW_VICTOR = 2;
    public static final int ELEVATOR_TOP_BUTTON_ONE = 8;
    public static final int ELEVATOR_TOP_BUTTON_TWO = 9;
    public static final int ELEVATOR_BOTTOM_BUTTON_ONE = 6;
    public static final int ELEVATOR_BOTTOM_BUTTON_TWO = 7;
    public static final int AUTO_SWITCH_ONE = 0;
    public static final int AUTO_SWITCH_TWO = 1;
    public static final int[] RATCHET_PISTON_CHANNELS = {1,3};
}
```



# PROGRAM FLOW

# HIERARCHY

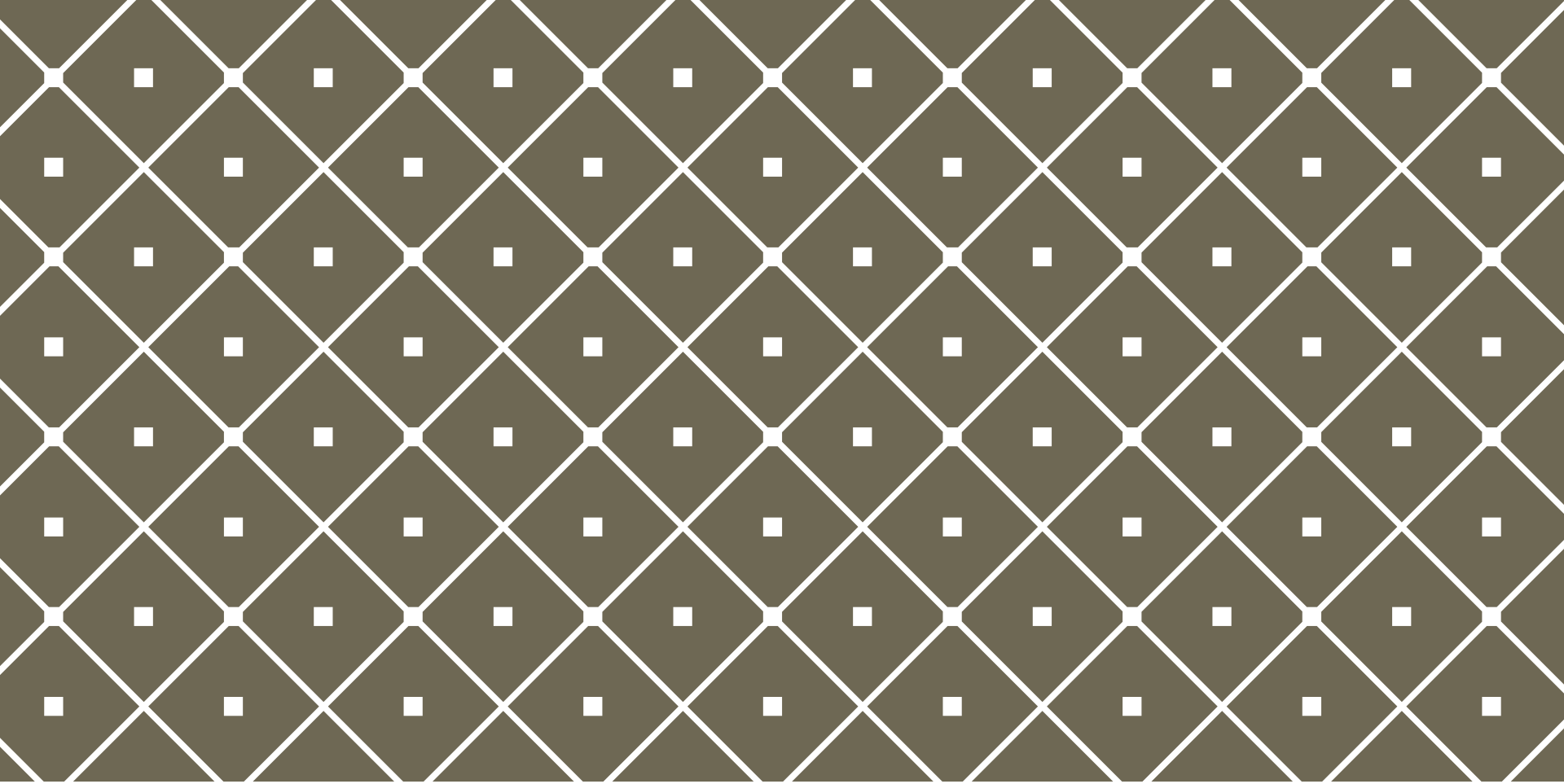

Robot.java
OI.java
Game Methods
Subsystems
Hardware Objects
Hardware Methods
Commands
Repeated Action
End Condition

# MORE ABOUT FLOW

We build our robot using **command-based** programming. Every action is performed using a command. This allows us to write less code for commonly used actions.

Remember:

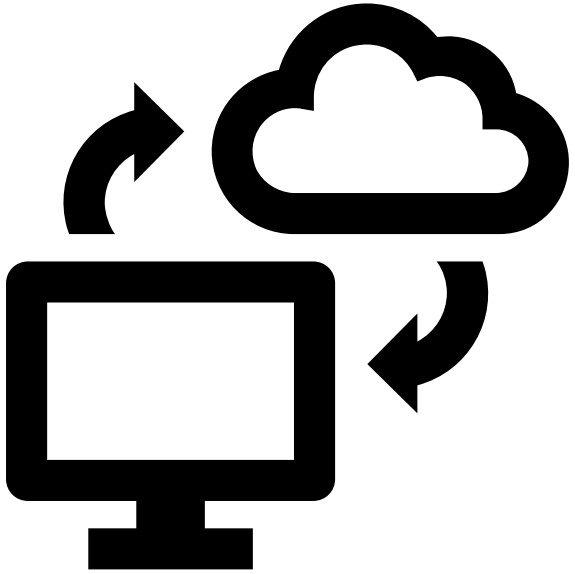
- All commands must have an **execute block** and a **finish condition**
- Commands should be **small** and **modular,** do not try to cram many tasks into one command
- Commands should declare what **subsystem** they are using



# GIT: A SHORT INTRODUCTION TO VERSION CONTROL

# WHAT IS GIT?

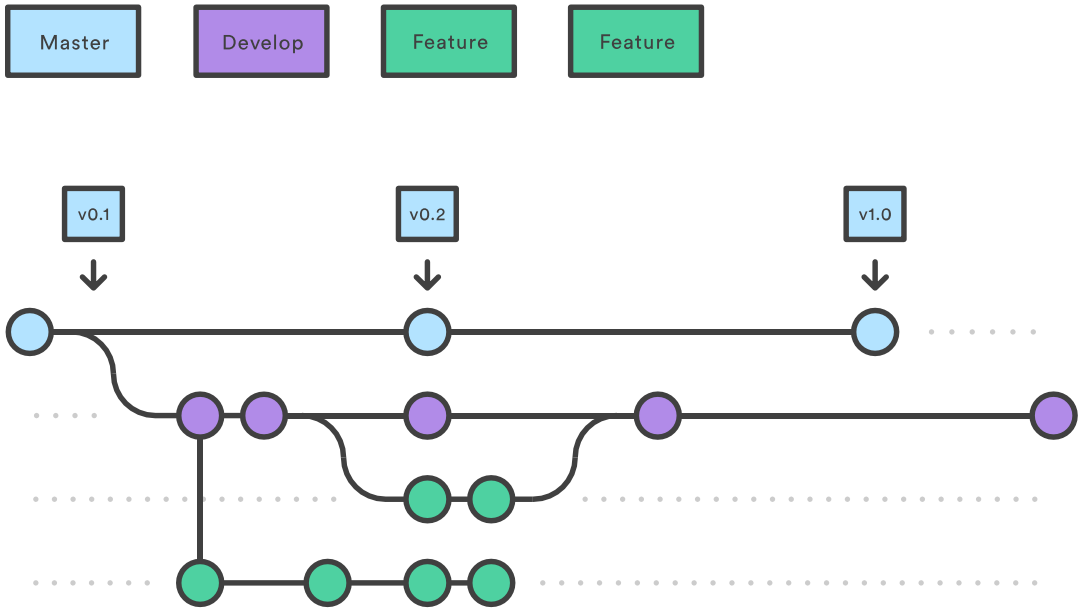
With multiple programmers working on the same program, there are bound to be differing versions. Git helps us manage these versions by providing a centralized **repository** where our code is stored. It keeps track of who changes what with **commits,** and can contain multiple versions of the same program on different **branches.** Git allows us to collaborate without having to worry as much about changing things.



# GIT FLOW

Important:

- Each dot is a **commit**
- The **master** branch always must have 100% working code
- Any changes should be made on **feature** branches of the **develop** branch



# IMPORTANT GIT COMMANDS

`git commit -avvv`

- When you are satisfied with the changes you have made, use this to create a new **commit**
- It will open up an editor where you can type a **commit message** detailing your changes

`git push`

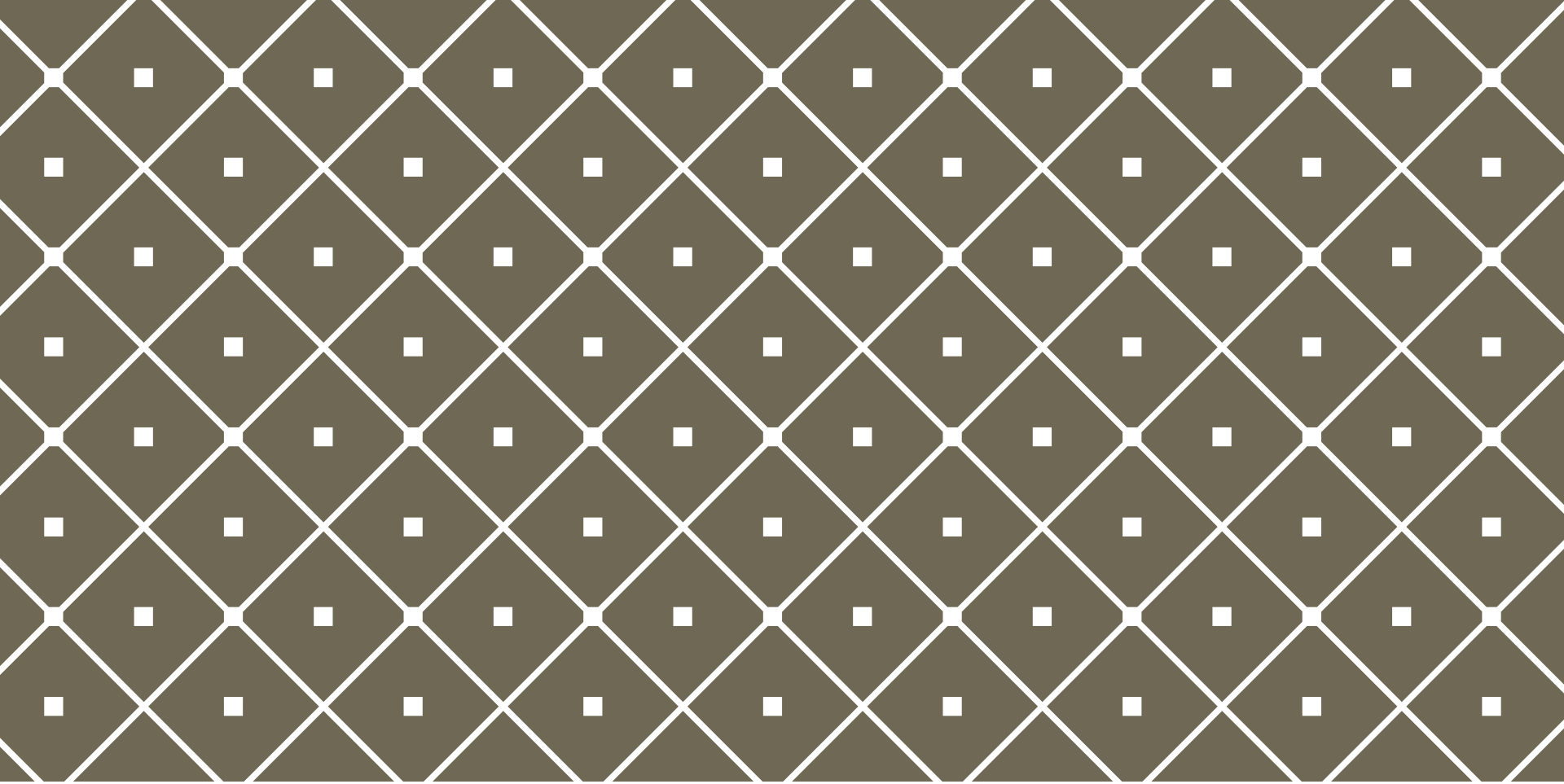
- This sends your new **commit** to the online central **repository,** where everyone can access it

`git branch branch-name`

- Use this command to create a new **branch** with an appropriate name

`git checkout branch-name`

- This command switches the code you are working on to the specified **branch**



# SOME THOUGHTS ON VISION PROCESSING

# WHAT IS VISION PROCESSING?

Using a camera on the robot, we can write code to analyze the robot's surroundings and perform appropriate actions. It would be most useful in autonomous, but could also assist the drivers in teleop.

Pros:

- Help determine actions we can't necessarily plan beforehand
- Allow robot to perform actions more precisely
- Looks good in alliance selection

Cons:

- Really hard
- We've never done it before

# 3M RETROREFLECTIVE TAPE

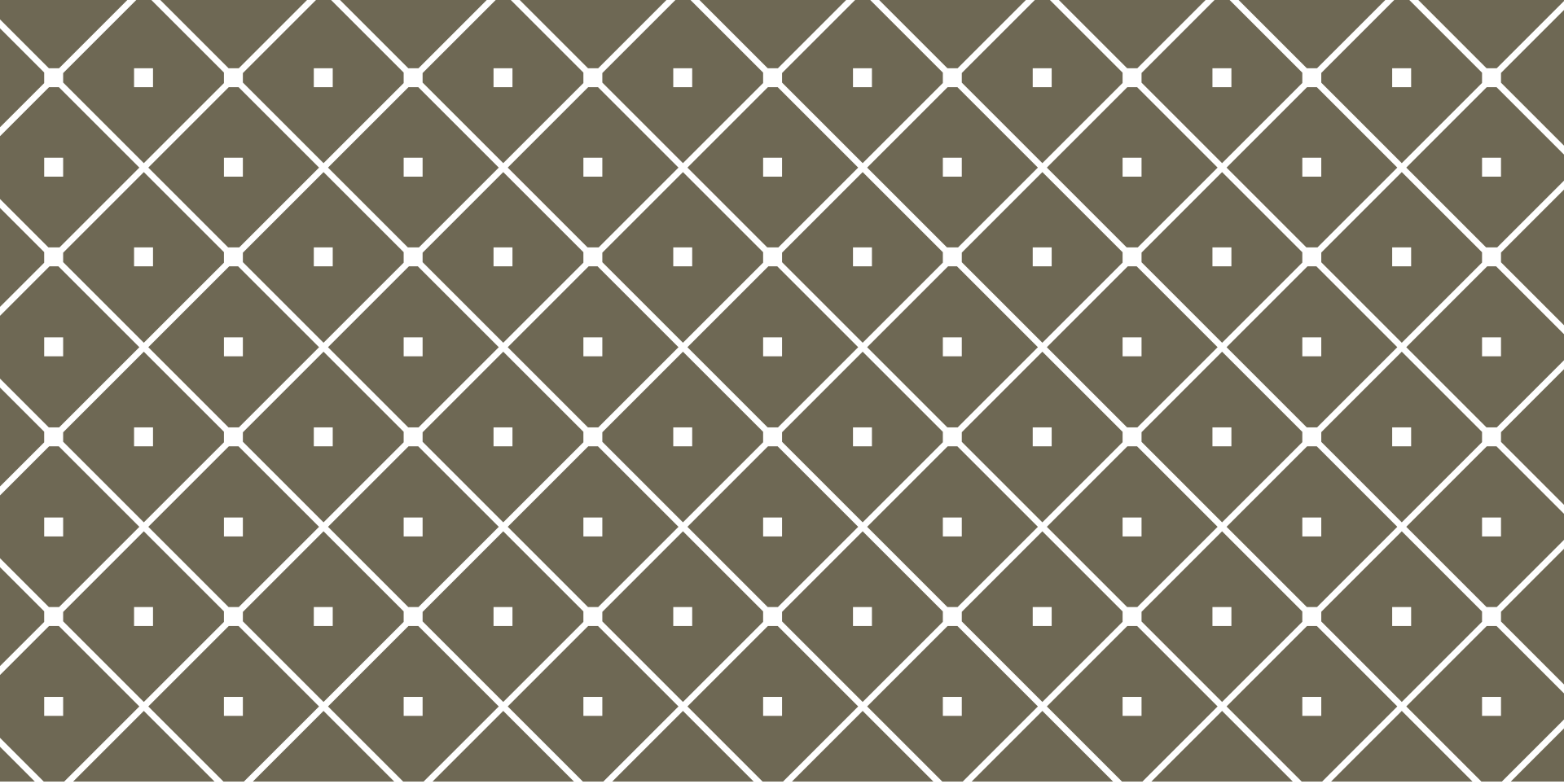

This special tape is what vision processing mostly relies upon. The field is marked with this tape in certain locations, and by pointing the camera at it, we can easily pick it out because it is so reflective. This allows us to have a very specific point of reference to control the robot based upon.

# NVIDIA TEGRA

We have a **coprocessor** called an NVIDIA Tegra which we can use for vision processing. It is designed to be very good at real-time image processing. We could use it on the robot to have extra power to do more complex vision processing tasks.

# VISION PROCESSING PROGRAMMING

The good thing about vision processing is that we do not need to use one specific programming language for it. We would most likely be using a software library called **OpenCV**, a complex but very efficient group of image processing tools. This library can be used in many different programming languages, but it would require quite a bit of work to figure out. Because of its complexity, vision processing programming would probably have a separate team working on it outside the normal robot programming.



# GRADLE BUILD SYSTEM

# CONNECTING TO THE ROBOT

We use a program called **gradle** to deploy code we write to the robot

There are 3 ways to connect to the robot

- USB Cable
- Ethernet Cable
- Wifi

We usually use the usb cable as it is the most reliable

The robot must be connected in some way in order to deploy code

# IMPORTANT GRADLE COMMANDS

`./gradlew build`

- Builds the code without deploying it, used to check if the code you are writing is correct

`./gradlew deploy`

- This command runs the build command, and then pushes the compiled code to the robot. The computer must be connected to the robot for this to work.