Who am I?

Brad Miller
brad@wpi.edu

- **Associate Director Robotics Resource Center**
- **Responsible for the WPILib**
  - **Used for LabVIEW, C++, and now Java**
- **Involved with FIRST since 2001**
- **One of the mentors for Team 190**

WPI

the posse

Alex Henning
Paul Malmsten

Robotics Engineering at WPI

- 3rd largest major on campus based on incoming freshman
- Students learn Mechanical Engineering, Software Engineering, and Electrical Engineering

- I used to say, “It’s not about the robots”. Now it is!

WPI

Introduction

C++ for FRC

- First language developed for FRC
- Full C++ implementation from Wind River using Workbench (eclipse)
- Fastest program possible
- Library makes robot programming easy

WPI

Java for FRC

- Provide a safe text based language
- Reduce the barrier to entry
- Have a program environment that matches what students learn in school
- Simplify robot programming even more

WPILibJ

- Equivalent functionality as C++/LabVIEW
  - Teams can learn about and use object oriented programming techniques
- Framework to support competition programs
- Runs on top of VxWorks
  - Full support for Java threading features
- Parity between C/C++ and LabVIEW libraries

Objects represent real world things

- Things like:
  - Motors
  - Sensors
  - The arm on your robot
  - Driver station
- What do all these have in common?
  - Associated data and state
  - Stuff you can ask the thing to do
- Object Oriented Programming is all about creating the right set of objects to model your robot

WPI

Classes

- **Classes represent *templates* of information**
  - **Like blank forms - just structure, no data**

- **Objects are filled out forms**
  - **Filling out the template (blank form) *instantiates* it**

WPI

Classes

- **Inside classes you put 3 things:**
  1. Variable definitions - what data the class knows about
  2. Methods - named pieces of code that represent what the class can do
  3. Constructors - the code that initializes the class when a new instance is created

- **Classes can extend other classes where they inherit everything the other class then add their own data and methods**

WPI

Robot Definition

```
package edu.wpi.first.wpilibj.templates;

import edu.wpi.first.wpilibj.SimpleRobot;

public class Team190Robot extends SimpleRobot {
    public void autonomous() {
      // autonomous code goes here
    }
    public void operatorControl() {
      // teleop code goes here
    }
}
```

This makes a class called Team190 robot that inherits all of the SimpleRobot capabilities and adds some of its own, such as specific autonomous and operatorControl behavior

Declarations & Initialization

```
RobotDrive drive;
Joystick leftStick;
Joystick rightStick;

// Constructor
public Team190Robot() {
        drive = new RobotDrive(1, 2);
        leftStick = new Joystick(1);
        rightStick = new Joystick(2);
}
```

This is the constructor (you know that because the name is the same as the name of the class - Team190Robot) and it's run when the class is instantiated.

In this case it happens to initialize the robot drive and 2 joysticks.

Autonomous part of program

```
public void autonomous(void)
{
    drive.drive(0.5, 0.0);
    Wait(2.0);
    drive.drive(0.0, 0.0);
}
```

Use the RobotDrive object drive method to go 0.5 speed (forward, half speed) with no turn then stop.

KEY TO THE EXAMPLE:

drive.drive(*speed*, *curve*)

speed: a value from -1.0 to 1.0 where 0.0 is stopped

curve: a value from -1.0 to 1.0 where 0.0 is no turn

TeleOp part of program

```
public void operatorControl()
{
    while (true)
    {
        drive.tankDrive(leftStick, rightStick);
        Timer.delay(0.01);
    }
}
```

Re-enable the watchdog since we aren't waiting any more

Use the RobotBase object tankDrive method to do arcade driving using our Joystick object

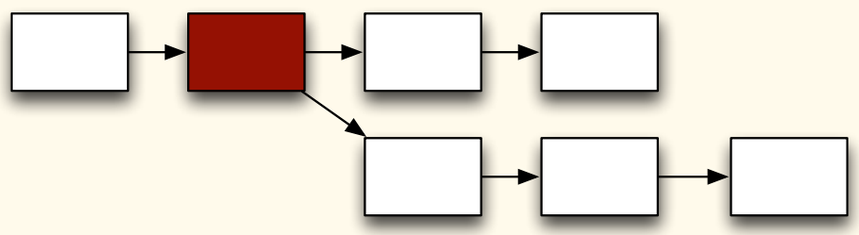
Organizing more complex progams

2012 WPILib*

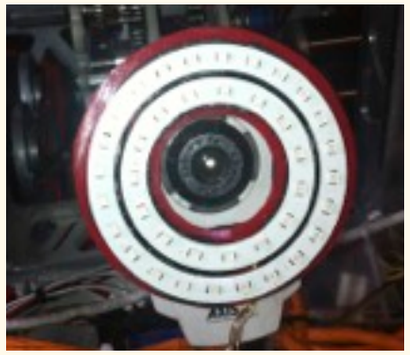
New features that add up to more than the sum of the parts

*These features available tomorrow at noon

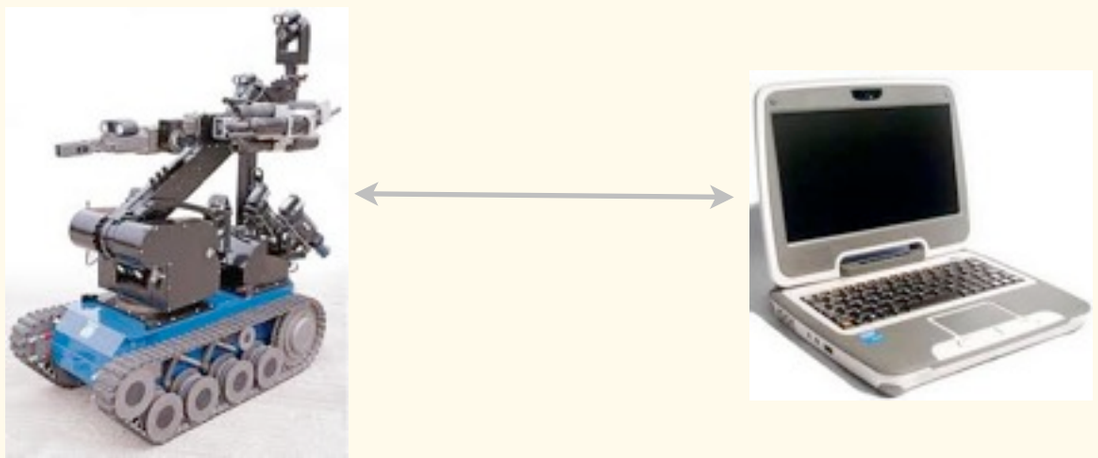
What's New?



Command-based programming model



Distributed vision processing



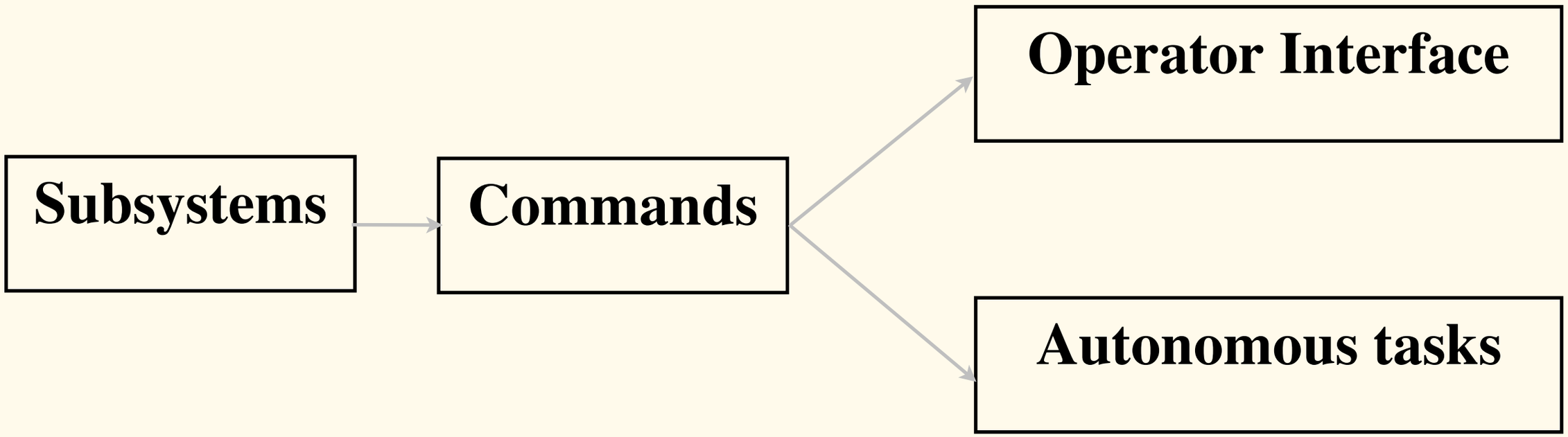
Smart dashboard
and Network tables

Command-based model

- Cookbook approach to writing robot programs
  - Easier to write
  - Less likely to get stuck as the program gets more complex
- Start writing programs as soon as the high level design is complete
  - Programmers don’t have to wait for a robot

WPI

The pieces


Operator Interface
Subsystems
Commands
Autonomous tasks

Subsystems

- **Major systems on the robot**
  - A class for every subsystem
  - List things that each subsystem:
    - can do (methods)
    - and has (sensors, motors, etc. as variables)

WPI

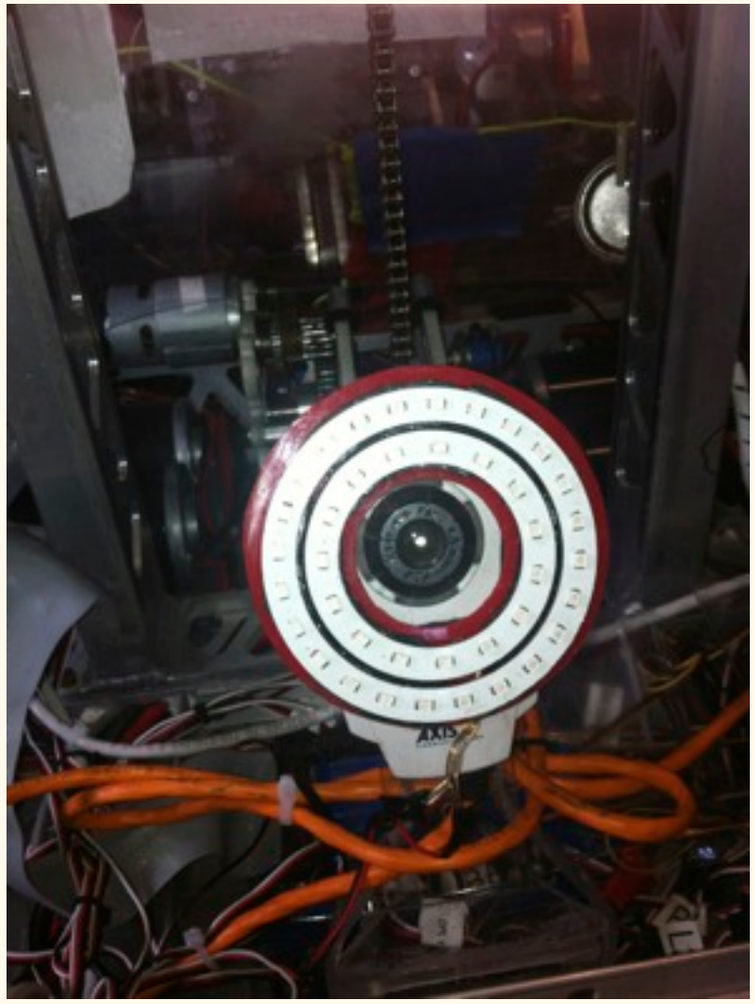
Vision processing

Can do these things:

- Find target x, y, angle
- Turn lights on
- Turn lights off

Has these things:

- Camera
- Light control



Shooter

Can do these things:

- Shoot balls
- Move balls to top
- Shoot to lower goal

Has these things:

- Ball collector motor
- Ball shooter motor
- Bin full sensor

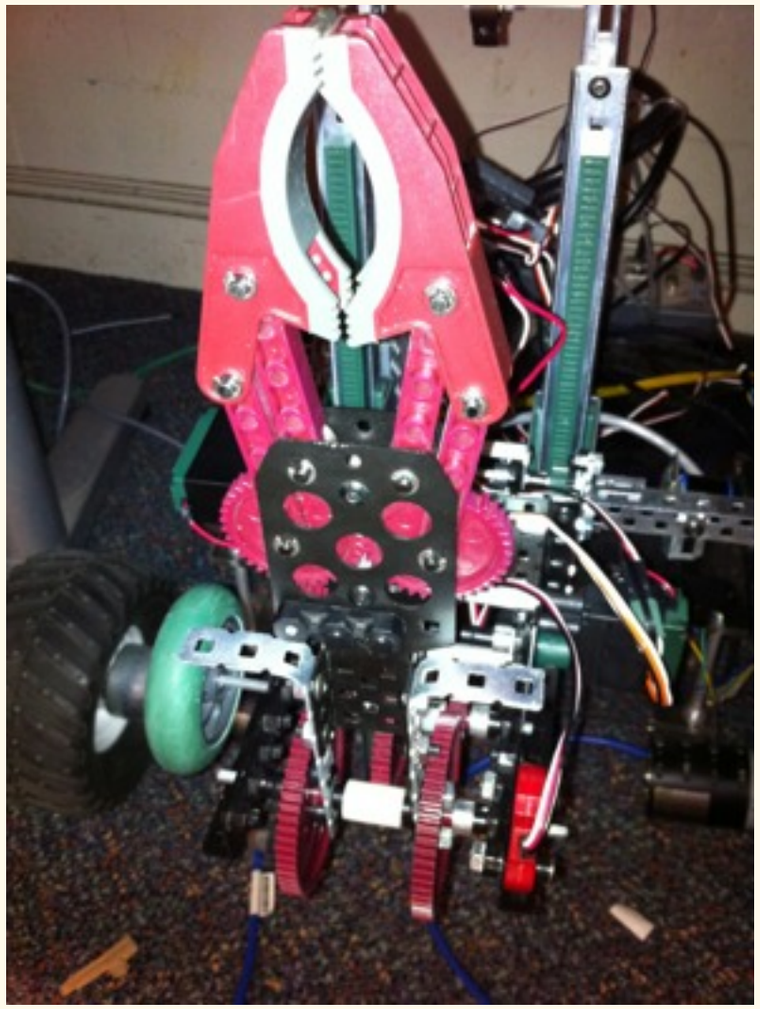
Grippers

Can do these things:

- Grip
- Release
- Raise up

Has these things:

- Tube sensor
- Solenoid actuator



Wrist

Can do these things:

- Move to angle
- Jog up/down

Has these things:

- Wrist motor
- Angle potentiometer

Elevator

Can do these things:

- Move to bottom
- Move to middle
- Move to top
- Jog up
- Jog down

Has these things:

- Drive motor
- Height potentiometer
- Upper limit switch
- Lower limit switch

Collectors

Can do these things:

- Rotate until empty
- Collect
- Move to position
- Uncollect
- Score tube

Creating a Subsystem

```
public class Claw extends Subsystem {
    Victor motor;

    public Claw() {
        motor = new Victor(7);
    }

    public void initDefaultCommand() {
        setDefaultCommand(new ClawDoNothing());
    }

    public void move(double speed) {
        motor.set(speed);
    }
}
```

Class should extend Subsystem

Initialize subsystem with the constructor

Optionally, set the default command that should run when no other command is controlling this subsystem

Methods that provide the capability of the subsystem

A little control theory

- **Use feedback for controlling a motor**
  - **You know what the mechanism *should do***
  - **Sensors tell what the mechanism *is doing***
  - **The difference is called the "error"**
- **The goal is to minimize the error**

- **PID is the name of a technique to control the error and PID is built into WPILib**

PID Control

- **PID has three different components that control the actuator**
  - **Proportional**
  - **Integral**
  - **Differential**

PIDSubsystem

```
public class Wrist extends PIDSubsystem {

    private static final double Kp = -2;
    private static final double Ki = 0.0;
    private static final double Kd = 0.0;

    public static final double PICKUP = 4.3,
            STOW = 2.5;

    Victor motor;
    AnalogChannel pot;
```

Just like a normal subsystem, but has an integrated PID controller object.

PID constants

Set some constants for the bounds/positions of the wrist.

Create a motor and a potentiometer (analog input)

PIDSubsystem

```
public Wrist() {
    super("Wrist", Kp, Ki, Kd);
    motor = new Victor(5);
    pot = new AnalogChannel(2);

    setSetpoint(STOW);
    enable();
}
```

Wrist constructor sets up PID constants and the initial set point. Then enable the PID Controller.

```
protected double returnPIDInput() {
    return pot.getVoltage();
}

protected void usePIDOutput(double output) {
    motor.set(output);
}
```

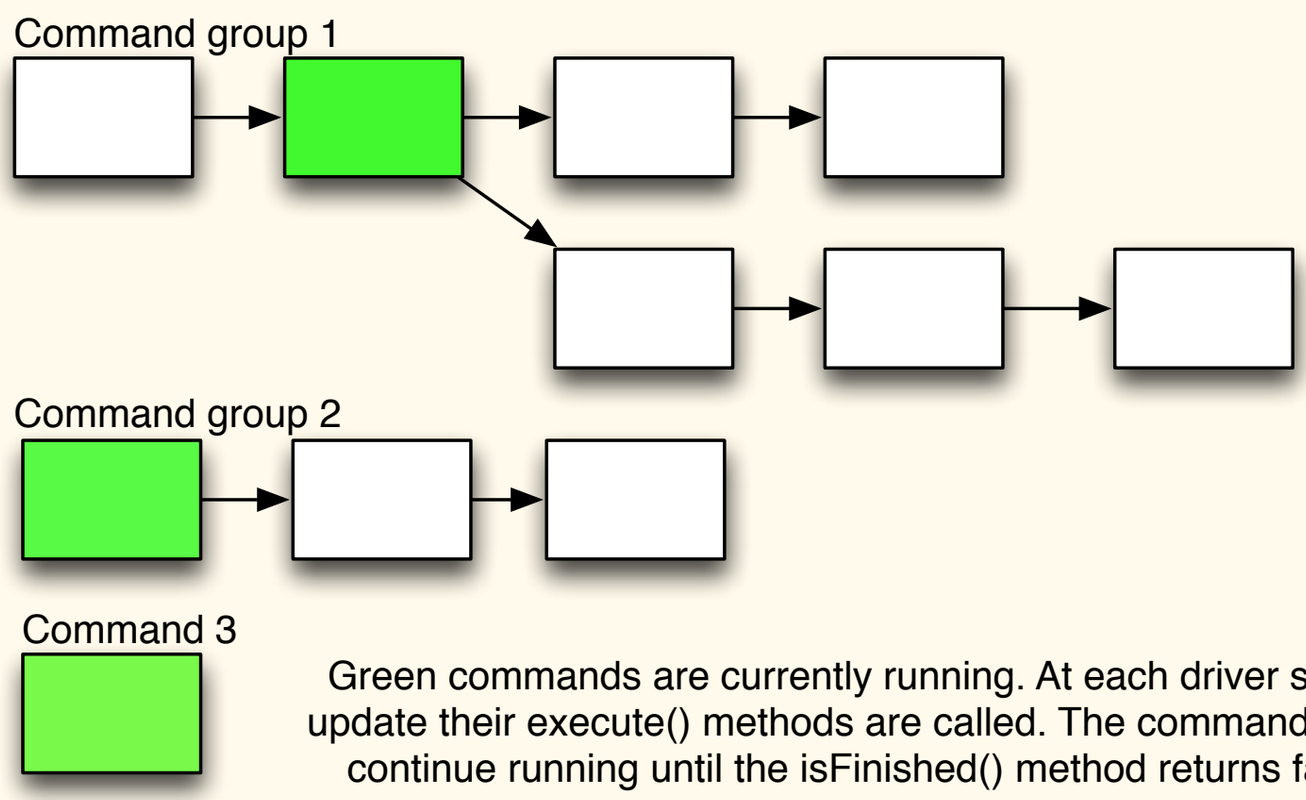
The PIDInput method gets the input sensor value that controls the device.

The PIDOutput method sets the actuator based on the computed output value.

WPI

Commands

- Robot actions that run over time
- Each command is a class based on Command
- Consists of a single or groups of actions
- Can be tested individually



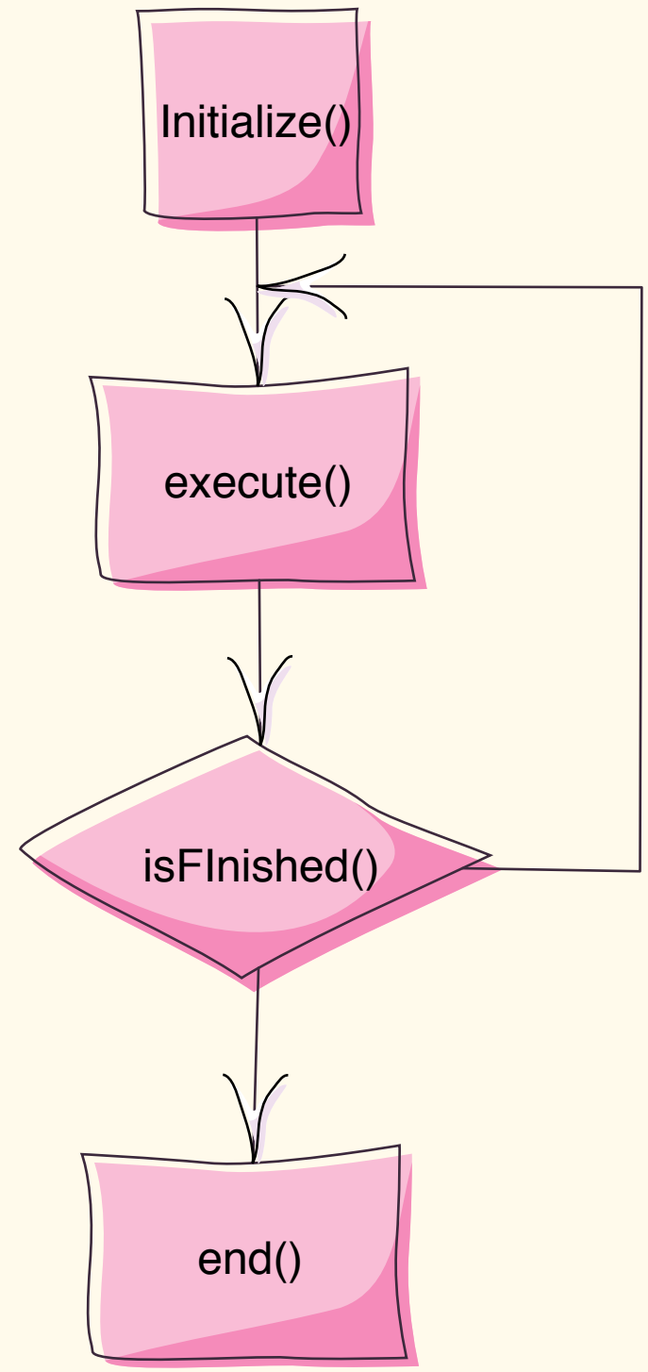
Green commands are currently running. At each driver station update their execute() methods are called. The commands each continue running until the isFinished() method returns false.

Command model

initialize() - initializes the command

execute() - does the command operation

isFinished() - returns true if finished



Happens every driver station update (20ms)

Creating a command

```
public class SetWristSetpoint extends CommandBase {
    private double setpoint;

    public SetWristSetpoint(double setpoint) {
        requires(wrist);
        this.setpoint = setpoint;
    }

    protected void initialize() {
        wrist.setSetpoint(setpoint);
    }

    protected void execute() { }

    protected boolean isFinished() {
        return Math.abs(wrist.getPosition() - setpoint) < .08;
    }

    protected void end() { }

    protected void interrupted() { }
}
```

As before... boilerplate code is provided for you when creating the Command class.

Constructor sets up the command class.

Initialize is called once when the command is scheduled then not again.

Execute is called repeatedly until the command is finished.

isFinished returns true when the command has completed whatever it is supposed to do.

Creating a Command

```
public class SetWristSetpoint extends CommandBase {
    private double setpoint;

    public SetWristSetpoint(double setpoint) {
        requires(wrist);
        this.setpoint = setpoint;
    }

    protected void initialize() {
        wrist.setSetpoint(setpoint);
    }

    protected void execute() { }

    protected boolean isFinished() {
        return Math.abs(wrist.getPosition() - setpoint) < .08;
    }

    protected void end() { }

    protected void interrupted() { }
}
```

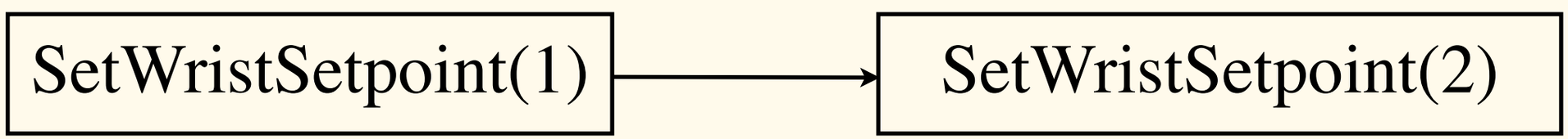
Command that sets the wrist to a particular angle (setpoint)

The setpoint is set once in the initialize() method, then it goes by itself

The command is finished when the wrist has reached the desired setpoint

Requirements

- **Commands can require (reserve) subsystems**
- **Newly scheduled commands on running subsystems replace older commands**



Both of these commands reserve the elevator subsystem

If the wrist is still moving up and a move down request is scheduled, it starts going down right away (except if non-interruptible)

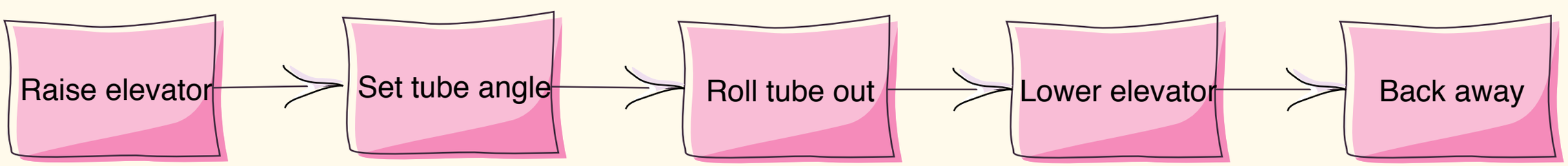
Command requirements

```
public SetWristSetpoint(double setpoint) {
        requires(wrist);
        this.setpoint = setpoint;
}
```

This command requires the wrist and will stop running when another command requiring the wrist is scheduled.

Groups

- Can be grouped to execute one after another
- Requirements of groups are the sum of the individual command requirements



Groups

```
public class SodaDelivery extends CommandGroup {

    public SodaDelivery() {
        addSequential(new PrepareToGrab());
        addSequential(new Grab());
        addSequential(new DriveToDistance(.11));
        addSequential(new PlaceSoda());
        addSequential(new DriveToDistance(.2));
        addSequential(new Stow());
    }
}
```

This group delivers a soda can by executing many commands in sequence. The individual commands each run until each is finished (isFinished() == true). The requirements for this command group are the union of all the requirements of all the commands that make it up.

Default Commands

- **Can be associated with a subsystem**
- **Runs whenever no other command is currently requiring that subsystem**

For example: using Joysticks to drive the robot whenever nothing else is using it (like some augmented or autonomous drive routines)

WPI

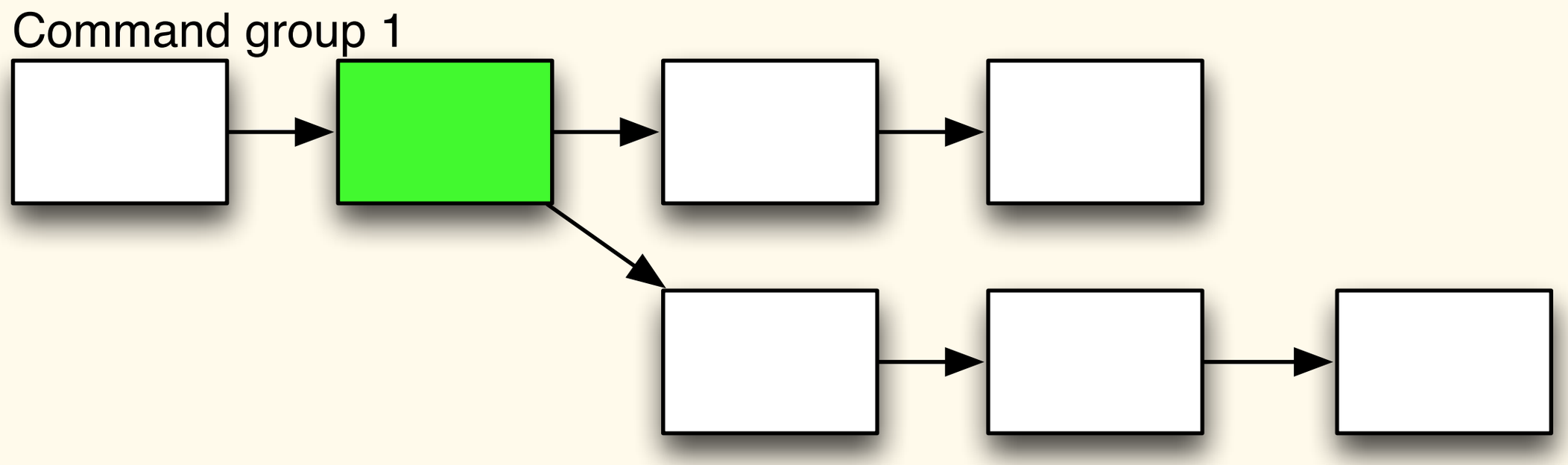
Timeouts

- **Maximum times for a command to run**
- **Complete based on either the command finishing its task or time running out**

Run a motor for a period of time to make sure that a mechanism has closed or moved into a set position, then stop.

Parallel Commands

- A command can start a parallel command
- Both run at the "same time" to get easy simultaneous operations



Operator Interfaces

- **Associate OI buttons with commands and groups**
- **Buttons can be:**
  - **User defined**
  - **Joystick**
  - **enhanced I/O module digital or analog inputs**

Operator Interfaces

```
Button armDownButton = new JoystickButton(stick, 4);
Button armUpButton = new JoystickButton(stick, 2);
Button autoButton = new JoystickButton(stick, 3);
```

Create buttons

```
armUpButton.whileHeld(new JogArmUpCommand());
armDownButton.whileHeld(new JogArmDownCommand());
autoButton.whenPressed(new DriveAndMoveArm());
```

Connect to OI

This is all you have to do to connect your buttons on the operator interface to the commands in the program!

Autonomous Tasks

- Use the same commands and groups shared between autonomous tasks and the operator interface
- Code to easily select from several autonomous commands to run using the driver station

More tools to help programmers

Network Tables

- **Distribute the program between the robot and the driver station computer!**
  - Named values written on the laptop appear on the robot and vice versa
  - Updates happen over a socket connection

```
NetworkTable camera = NetworkTable.getTable("camera");

if (!camera.getBoolean("found")) {
    angle = 0;
    isSeen = false;
} else {
    distance = (2737 / ((double) camera.getInt("major"))) - 12;
    double width = distance * .45;
    double x = width * Math.abs(((double) camera.getInt("x")) / 320.0 - 1.0);
    angle = Math.toDegrees(MathUtils.atan(x / distance));
    if (camera.getInt("x") < 320) angle *= -1;
}
```

Smart Dashboard

- **Used to create, well, a dashboard for the robot**
- **On the robot:**
  - **Put named data values**
  - **Camera image (maybe with processing)**
  - **Put the command scheduler class instance**
  - **Put subsystems**
  - **PID tuning widget**
  - **and others...**

Smart Dashboard

- You can
  - Change the layout
  - Display camera images
  - Create extensions in Java as new widgets
  - See connection status
  - Value chooser for Autonomous programs
  - and more...
- Works with Java and C++

WPI

Smart Dashboard

- Excellent debugging tool and general robot status monitor
- User customizable layouts
- Values can be displayed in a variety of formats: numeric, dials, sliders, etc.
- Automatic, but you can save layout
- No work to set up

Smart Dashboard

```
// button to turn on camera ring light
Button button = new InternalButton();
SmartDashboard.putData("Shine Camera Light", button);

// Wrist jog up/down buttons that activate commands
SmartDashboard.putData("Jog Up", WRIST_JOG_UP);
SmartDashboard.putData("Jog Down", WRIST_JOG_DOWN);

SmartDashboard.putData("Collect", CLAW_COLLECT);
SmartDashboard.putData("Stop Collect", CLAW_STOP_COLLECT);

// Button to activate the WristJog command
SmartDashboard.putData("Test Command", new WristJog(true));
```

Distributed Vision Code

- **Distribute vision processing between driver station computer and robot**
  - Use either NI Vision or OpenCV libraries
  - We'll supply some experimental Java OpenCV wrappers
  - Use the Network Tables to easily get values back to the robot
  - Send images via HTTP connection through the robot access point / ethernet switch (places no load on the cRIO)

WPI

Distributed vision code

- Allows more complex vision algorithms
- Can run independently of the cRIO to minimize CPU load on the robot

WPI

Microsoft Kinect

- Very cool sensor from Microsoft
- Uses structured light to generate a depth map of the scene
- Software creates a skeleton of 2 operators

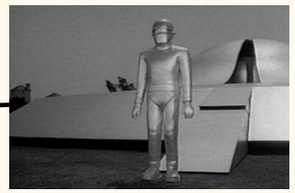
Kinect - Helping drive the robot

- **The idea:**
  - Use the skeleton to recognize gestures that operate the robot
- **The gesture interpretation can be done either on the robot or the driver station**
- **FIRST will supply a sample "Kinect Server" that interprets *standard* gestures**
  - delivered to the robot as joysticks

Gesture Interpretation

Kinect Server

Driver Station



- **Kinect server**
  - Connects to the Kinect and recognizes some gestures and delivers them to the DS along with the raw skeleton
  - Can be replaced by the team for custom processing
- **Driver station**
  - Sends Kinect, joystick, and other inputs to the robot
- **Robot**
  - Uses the *fake* joysticks or interprets the gestures

Documentation

- WPILib Cookbook
- Getting Started with Java
- Getting Started with C++
- WPILib Users Guide

- Includes JavaDocs for all APIs
- Java Tutorial (at java.sun.com)

- And user generated JavaDoc for your own classes