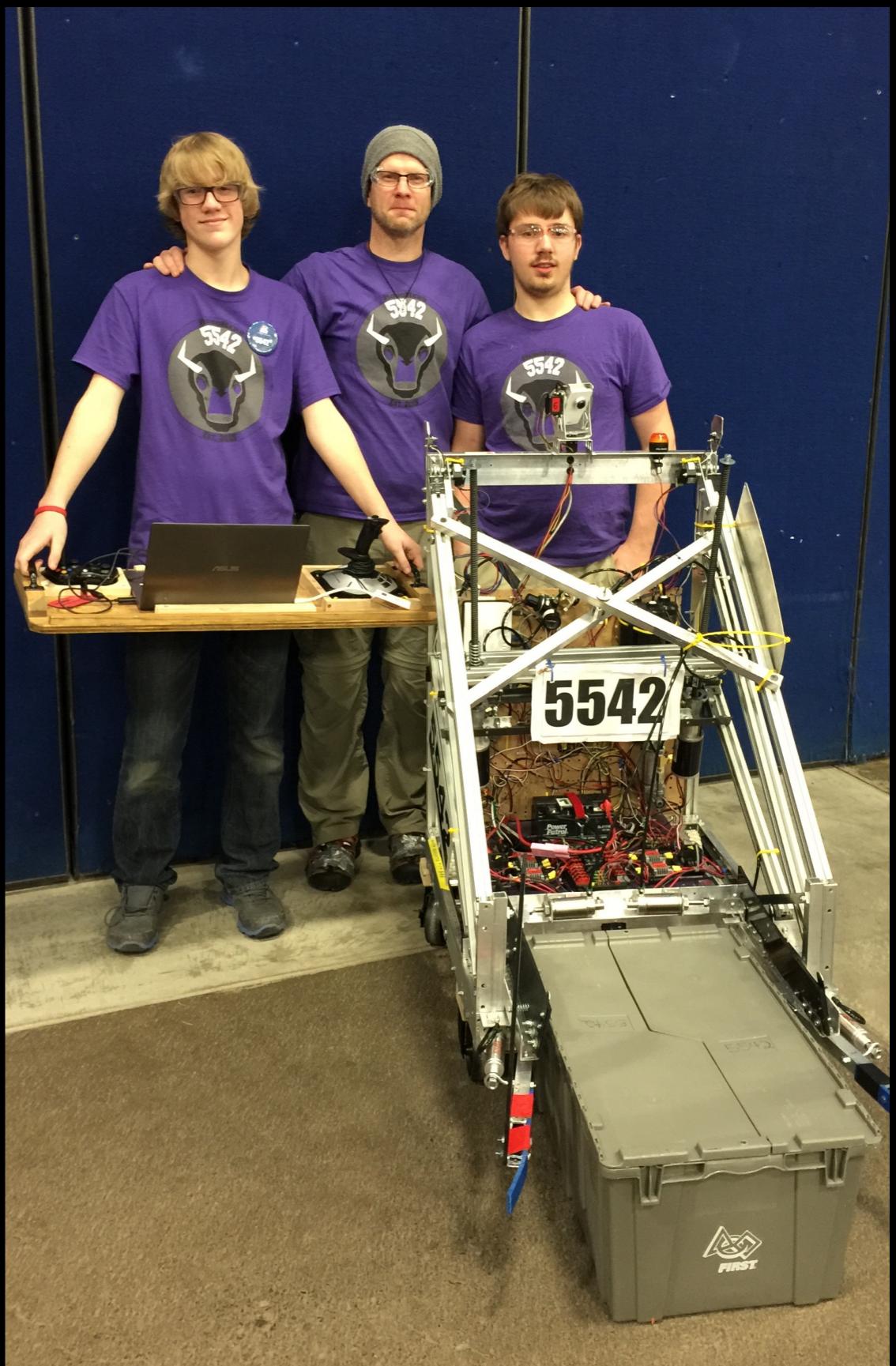


FRC Control System Basics

Coding Languages - C++ / Java

Chris Roadfeldt

chris@roadfeldt.com



Who am I and what do I know?

- Lead CSA
- CIS (4607) mentor.
- Enough to be dangerous... :)
- Proper research and planning saves time and frustration.
- Taking on a challenge brings growth.
- Perl, Bourne, some C, some C++, some Java, etc...
- Basic electronics, 3d printing, Unix, Linux
- General geekery



Topics

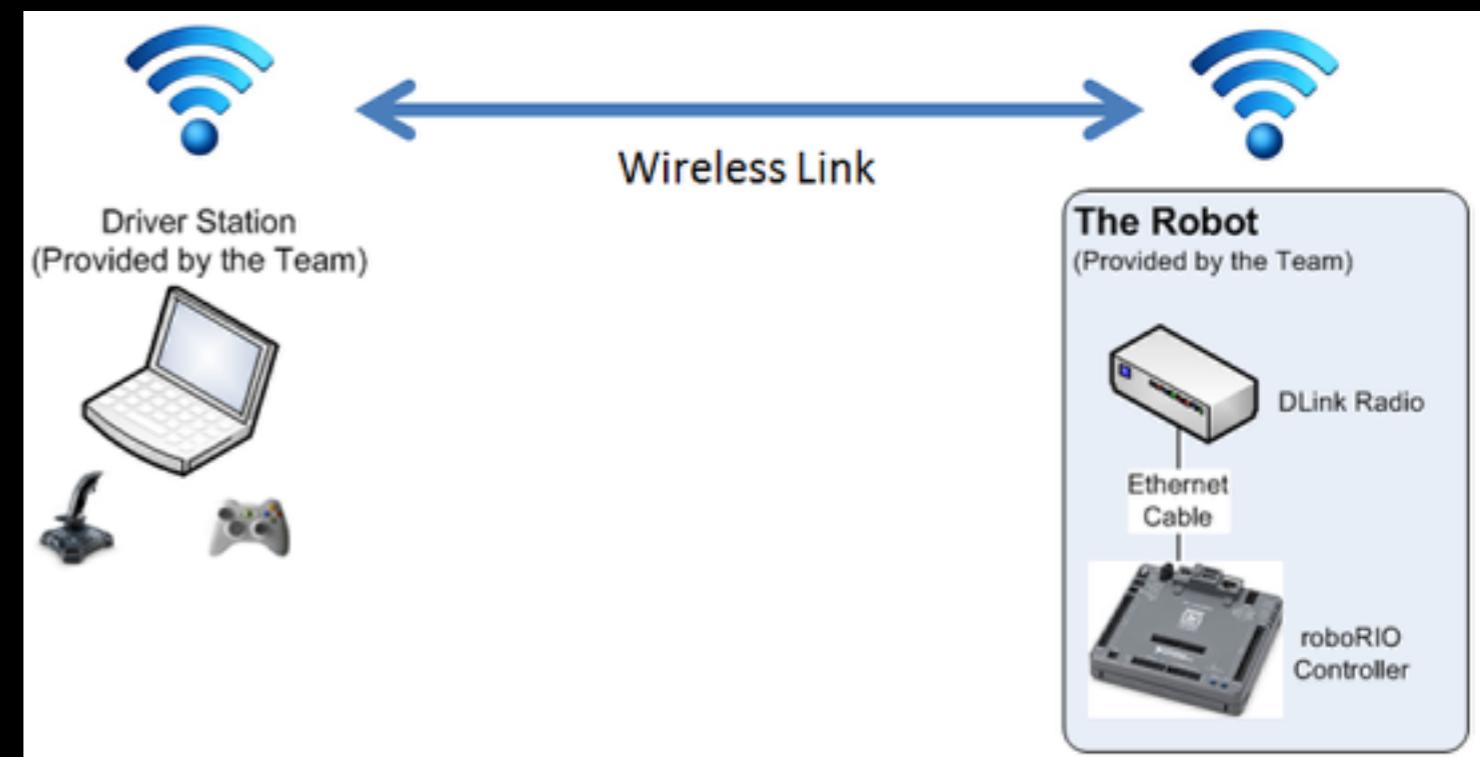
- Facilitate and promote discussion intra-team and inter-team.
- How do we control a robot?
 - Discuss flow of driver input to robot action and the components involved.
- Q & A
- If there is time.
 - What is WPILib?
 - Role of WPILib
 - WPILib Framework overview
 - WPILib base classes
 - Pros / Cons of C++ and Java
 - Source Control - GIT

Resources / Helping each other out

- <http://wpilib.screenstepslive.com/s/4485>
 - The resource for FRC control system documentation
- <https://www.reddit.com/r/FRC/>
 - Discussion group related to FRC
- <http://www.chiefdelphi.com/forums/portal.php>
 - Battle tested, well known site for all things geeky, robotics and FRC related.
- Can not emphasize enough the importance of reaching out to mentors, other teams or discussion groups.
- Never be afraid to ask your question.
- Ask your questions anytime during this discussion.

Driver Input to Robot Action

- Xbox / Joystick - AKA Operator Interface
- DS ->
- Laptop / OS ->
- Network ->
- RoboRio / OS->
- Robot Code / WPILib ->
- Stage appropriate code segment ->
- command ->
- subsystem ->
- WPILib HAL ->
- CAN / PWM / GPIO ->
- Motor controller

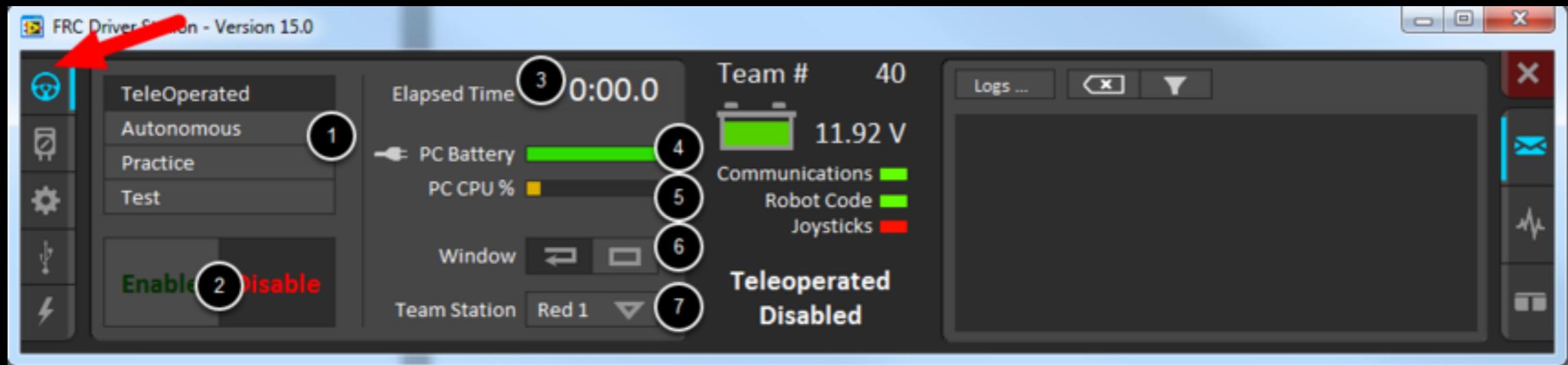


Operator Interface

- Takes input from Operator / Driver
 - Joystick, XBox controller, Arduino, etc...
- Sends input to Driver Station
- Can be any applicable device you want.
- Devices not in the kit of parts will need to be custom coded.
- Provides Feedback to Operator / Driver
 - SmartDashboard, ShuffleBoard - NEW!



Driver's Station



- Controls the Robot, only thing that controls the robot.
 - Sends driver input, dashboard inputs, etc
 - Captures and logs robot runtime information
 - Interfaces with FMS.
 - Polls Robot at regular intervals for health check, positive control confirmation and timing.
 - When in trouble, HIT THE SPACE BAR!

Role of FMS

- FMS controls
 - Field electronics
 - Network infrastructure
 - Match scheduling
 - All field Hardware
 - Timers, team lights, estops, etc
 - Scoring in real-time
- Audience screen
- Uploading results to internet
- FMS talks to each team's Driver Station
- FMS Does NOT control the robot directly
- FMS Whitepaper
 - <http://wpilib.screenstepslive.com/s/4485/m/24193/l/291972-fms-whitepaper>



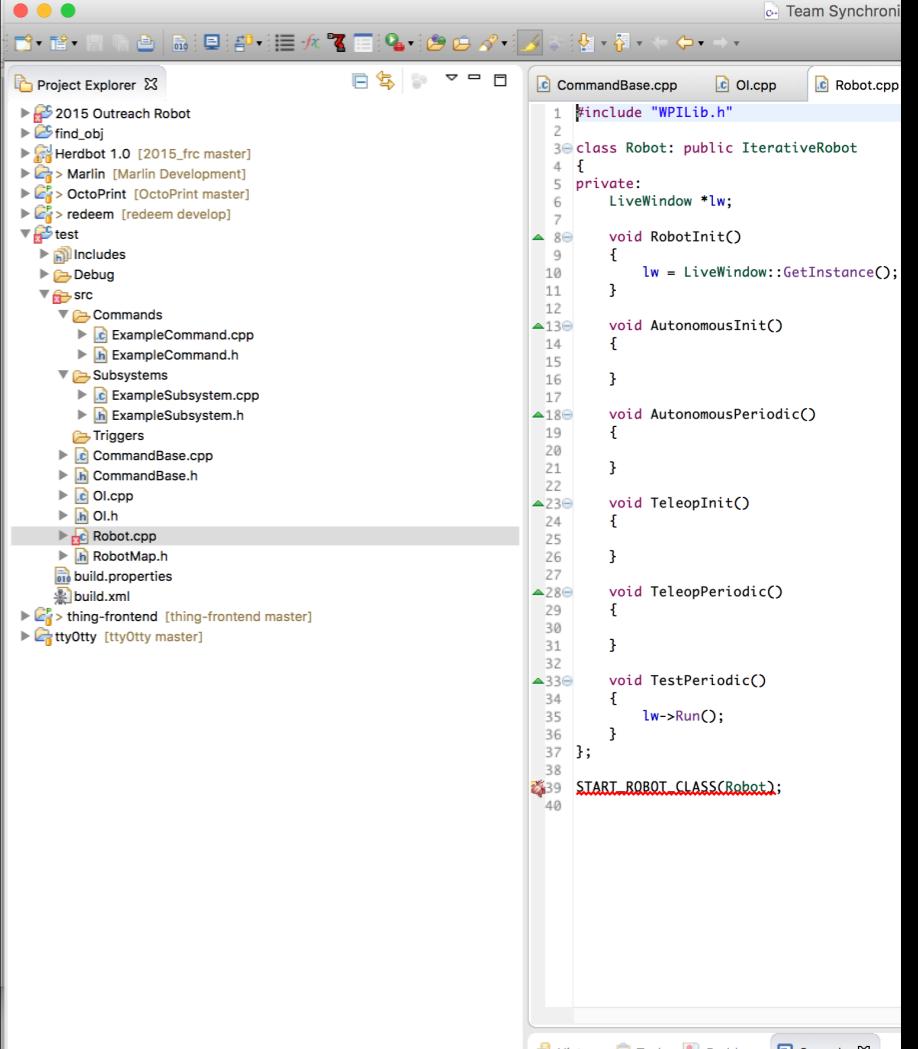
RoboRio / OS

- The Brains of the Robot
- Runs Linux
- Robot Code runs as app
- Provides several interface buses
 - CAN, I2C, PWM, DIO, SPI, MXP
- Controls everything directly or indirectly on Robot.



Robot Code

- Written using WPILib
- Officially supported languages
 - C++, Java, LabView - WPI
- Unofficial languages
 - Python - Not from WPI
 - Takes input from DS and uses RoboRio busses to control hardware devices.
 - Must be in control of all physical actions the robot can make.



The screenshot shows a software development environment with a "Project Explorer" window on the left and a "Code Editor" window on the right. The Project Explorer lists several projects and files, including "2015 Outreach Robot", "find_obj", "Herobot 1.0 [2015_frc master]", "Marlin [Marlin Development]", "OctoPrint [OctoPrint master]", "redeem [redeem develop]", and a "test" folder containing "Includes", "Debug", and "src" subfolders. The "src" folder contains subfolders for "Commands", "Subsystems", and "Triggers", along with files like "ExampleCommand.cpp", "ExampleCommand.h", "ExampleSubsystem.cpp", "ExampleSubsystem.h", "CommandBase.cpp", "CommandBase.h", "OI.cpp", "OI.h", "Robot.cpp", and "RobotMap.h". The "Code Editor" window displays the "Robot.cpp" file, which includes the header "#include <WPILib.h>" and defines a class "Robot" that inherits from "IterativeRobot". The code includes methods for "RobotInit()", "AutonomousInit()", "AutonomousPeriodic()", "TeleopInit()", "TeleopPeriodic()", "TestPeriodic()", and "START_ROBOT_CLASS(Robot);". The "Robot.cpp" file is highlighted in the editor.

```
#include "WPILib.h"
class Robot: public IterativeRobot
{
private:
    LiveWindow *lw;
    void RobotInit()
    {
        lw = LiveWindow::GetInstance();
    }
    void AutonomousInit()
    {
    }
    void AutonomousPeriodic()
    {
    }
    void TeleopInit()
    {
    }
    void TeleopPeriodic()
    {
    }
    void TestPeriodic()
    {
        lw->Run();
    }
};

START_ROBOT_CLASS(Robot);
```

What is WPILib?

- The WPI Robotics library (**WPILib**) is a set of software classes that interfaces with the hardware and software in your FRC robot's control system. There are classes to handle sensors, motor speed controllers, the driver station, and a number of other utility functions such as timing and field management.
- Generalized abstraction layer between your code and the hardware.
- Created by Worcester Polytechnic Institute Robotics Resource Center
- <http://wpilib.screenstepslive.com/s/4485/m/13809/l/241852-what-is-wplib>



WPI

Motor Controllers

- Controls electrical power sent to motors.
- Connected to RoboRio via a Bus. eg; CAN
- Takes input from Robot Code only.
- Connected to PDP for power source.
- Can be pro-active or reactive.
- Talon SRX / Victor SPX vs Talon SR / Victor 883



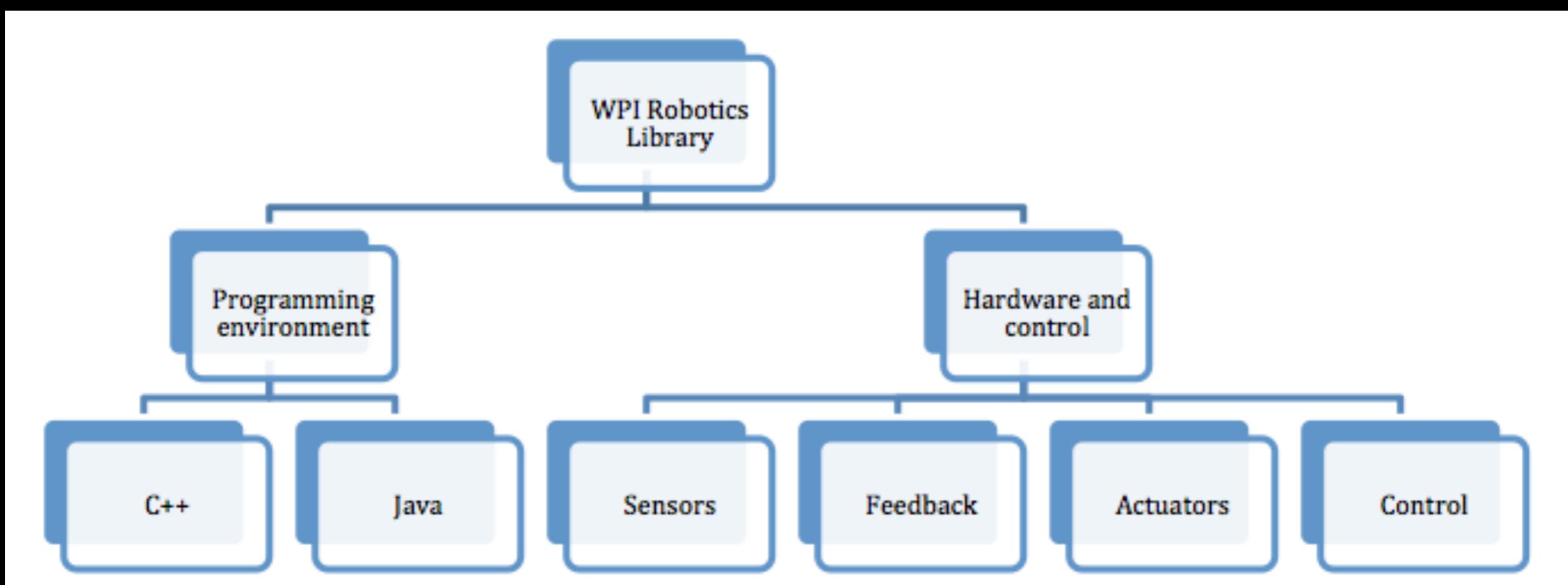
Q & A

Role of WPILib

- Provides a generalized common programming framework, API and HAL to control your robot.
- Provides templates for program logic flow.
 - AKA: Base Classes
- Provide generalized access to various hardware components.
 - Sensors, motor controllers, rangefinders, servos, CAN bus, I2C, GPIO, etc..
- Integrates with Driver's Station and Smart Dashboard
- Allows for code re-use.
 - WPILib is thoroughly tested code.
 - Carry over code from previous years.
 - Intra and Inter-team sharing of code
 - etc...

WPILib Framework

High level overview of components.



Base Classes

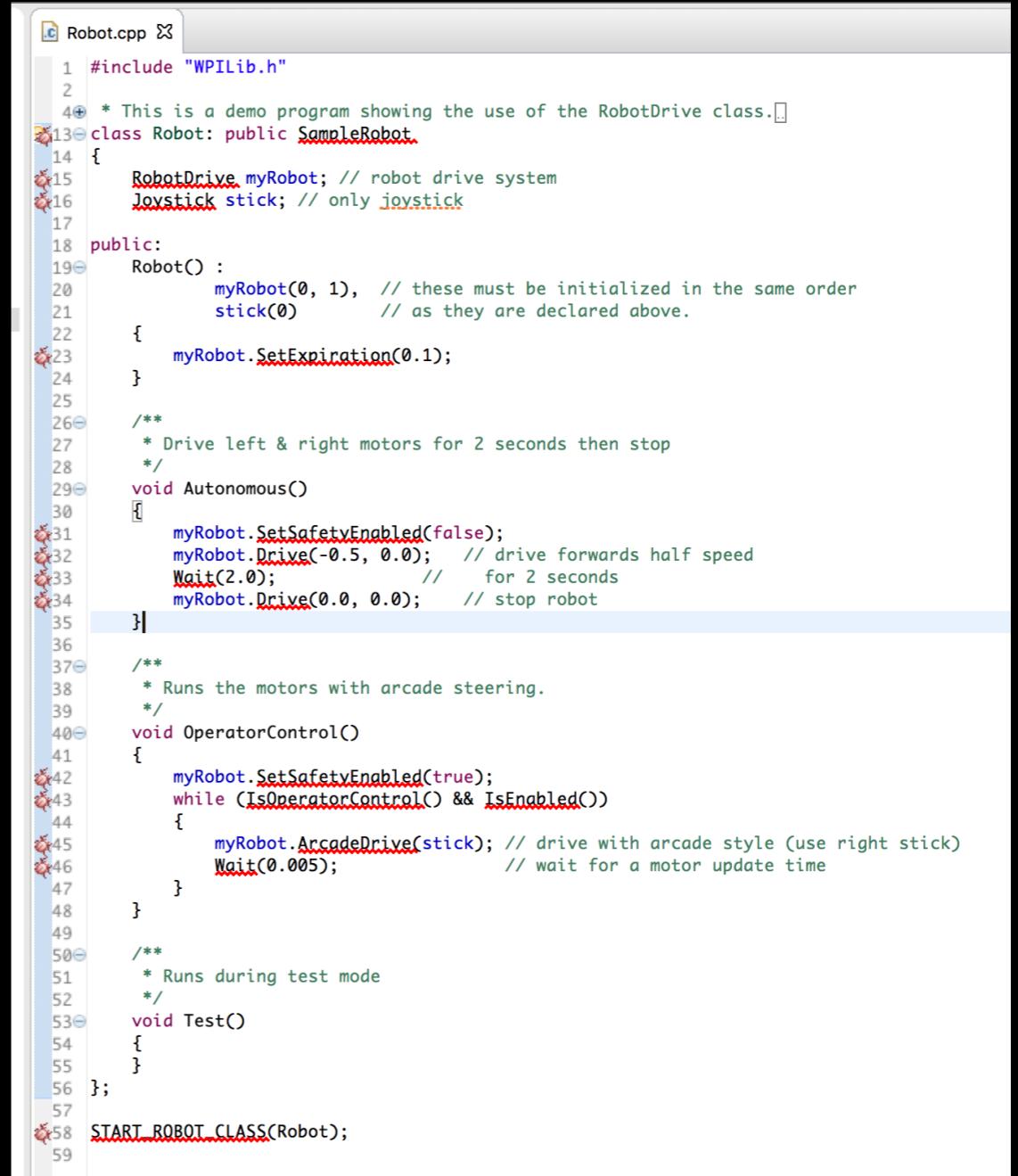
- Sample Robot
 - Allows for total control over program flow.
 - Code is called once per stage.
 - Extremely flexible
 - Learning curve depends on complexity.
- Iterative Robot
 - Provides a structure to build robot code from.
 - Minimal learning curve
 - Code is called in a periodic and iterative nature.

Base Classes

- Timed Robot
 - Based on Iterative.
 - Internally Timed - More resilient / robust robot actions that require timing.
 - Possible sync issues with controls. Breaks sync with DS.
- Command Based Robot
 - Provides a structure to build robot code from.
 - Moderate learning curve
 - Code is called in a periodic / iterative and branched based on stimuli.

Sample Robot - C++

- You control the logic flow.
- Provides very basic code to interface with robot.
- All functionality is in your code.
- Most logic flow is in your code.
- Useful for quick testing or very advanced logic flow.
- Two stages
 - Autonomous
 - TeleOp
- Requires loop to continue to function.
- Code readability depends on complexity.
- All code in one file.



A screenshot of a code editor window titled "Robot.cpp". The code is written in C++ and uses the WPILib library. It defines a class "Robot" that inherits from "SampleRobot". The class contains three methods: "Autonomous()", "OperatorControl()", and "Test()". The "Autonomous()" method initializes the robot drive system and drives it forward for 2 seconds. The "OperatorControl()" method runs the motors with arcade steering while the operator is controlling the robot. The "Test()" method is a placeholder for testing. The code also includes a call to "START_ROBOT_CLASS(Robot);".

```
#include "WPILib.h"

* This is a demo program showing the use of the RobotDrive class.

class Robot: public SampleRobot
{
    RobotDrive myRobot; // robot drive system
    Joystick stick; // only joystick

public:
    Robot() :
        myRobot(0, 1), // these must be initialized in the same order
        stick(0) // as they are declared above.
    {
        myRobot.SetExpiration(0.1);
    }

    /**
     * Drive left & right motors for 2 seconds then stop
     */
    void Autonomous()
    {
        myRobot.SetSafetyEnabled(false);
        myRobot.Drive(-0.5, 0.0); // drive forwards half speed
        Wait(2.0); // for 2 seconds
        myRobot.Drive(0.0, 0.0); // stop robot
    }

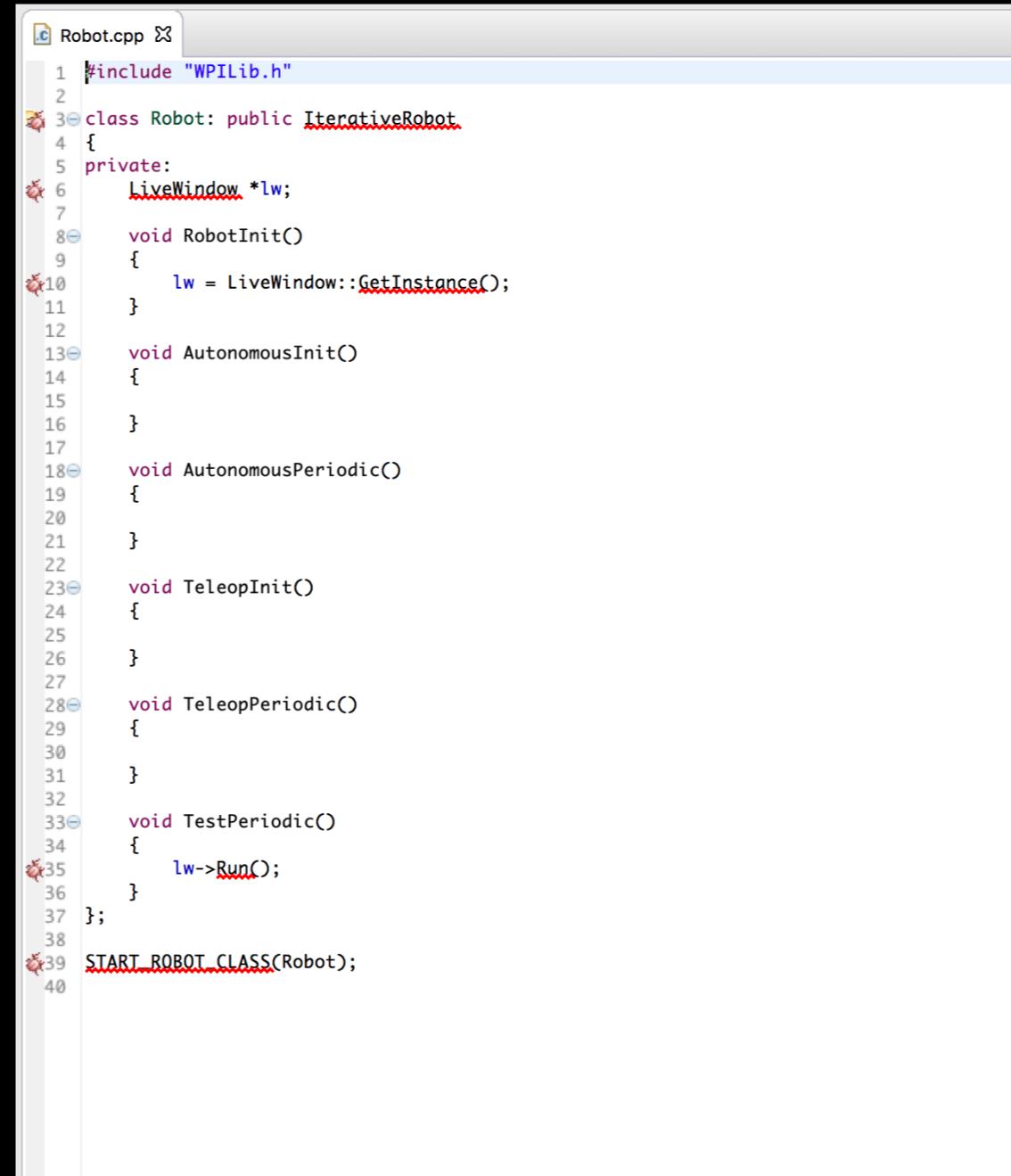
    /**
     * Runs the motors with arcade steering.
     */
    void OperatorControl()
    {
        myRobot.SetSafetyEnabled(true);
        while (IsOperatorControl() && IsEnabled())
        {
            myRobot.ArcadeDrive(stick); // drive with arcade style (use right stick)
            Wait(0.005); // wait for a motor update time
        }
    }

    /**
     * Runs during test mode
     */
    void Test()
    {
    };
};

START_ROBOT_CLASS(Robot);
```

Iterative Robot - C++

- You kinda control the logic flow.
- Provides basic logic flow.
- Robot control is mostly handled by your code.
- Four stages
 - Each stage of has two modes
 - Init - Called once at start of stage. Robot environment setup code goes here
 - Periodic - Called approximately every 20ms during stage. Robot control code goes here.
 - All code in periodic stages is run when called.
 - Good for basic operation with minimal complexity.
 - Code readability depends on complexity.
 - All code in one file.



```
#include "WPILib.h"

class Robot: public IterativeRobot
{
private:
    LiveWindow *lw;

void RobotInit()
{
    lw = LiveWindow::GetInstance();
}

void AutonomousInit()
{
}

void AutonomousPeriodic()
{
}

void TeleopInit()
{
}

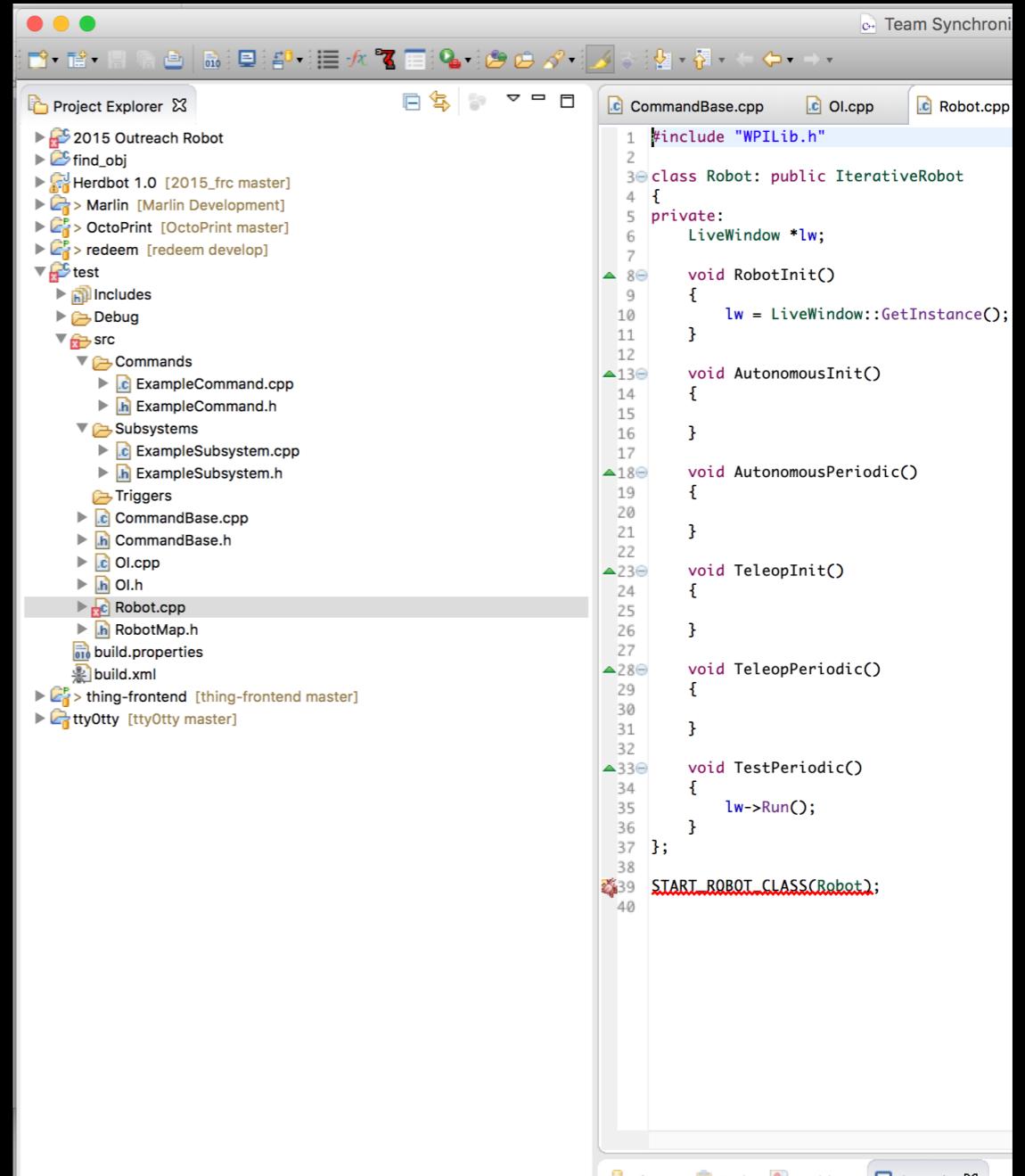
void TeleopPeriodic()
{
}

void TestPeriodic()
{
    lw->Run();
};

START_ROBOT_CLASS(Robot);
}
```

Command Based Robot - C++

- You split up and kinda control the logic flow.
- Based off of iterative.
- Good for any level of operation complexity.
- Code is broken up into separate files based on purpose.
- Code readability has a higher learning curve.
 - Might be overkill for simple robot operation.
 - Code is granularly scoped. Subjectively easier since each file has limited purpose.
- Code is broken up into small chunks based on action.
- More complex actions are built using those smaller actions.



The screenshot shows a C++ development environment with the following details:

- Project Explorer:** Displays the project structure under "2015 Outreach Robot". It includes several subfolders like "find_obj", "Herdbot 1.0", "Marlin", "OctoPrint", "redeem", "test", and "src". Within "src", there are subfolders "Commands", "Subsystems", "Triggers", and "Robot". The "Robot" folder contains "Robot.cpp" and "RobotMap.h".
- Code Editor:** Shows the content of "CommandBase.cpp". The code defines a class "Robot" that inherits from "IterativeRobot". It includes a private member "LiveWindow *lw;" and implements methods for "RobotInit()", "AutonomousInit()", "AutonomousPeriodic()", "TeleopInit()", "TeleopPeriodic()", and "TestPeriodic()". The "Robot" class also calls "START_ROBOT_CLASS(Robot);".

C++

- Pros

- Speed - In theory, compiled C++ code is faster than Java.
- Broad support - Numerous resources available for learning C++ on the net.
- Battle tested - C++ has existed in some form since 1979.
- Object orientated
- Similar in methodology to other object orientated languages - in part because other object orientated languages are strongly influenced by C++
- Ability to incorporate wide range of 3rd party libraries.
- Threading

- Cons

- Requires some level of programming knowledge
- Moderate learning curve if you are new to programming in general
- Memory Management.....
- Not graceful when it crashes - Can be mitigated with try / catch.
- Debugging robot code requires additional configuration and components.
 - <http://wpilib.screenstepslive.com/s/4485/m/13810/l/145321-debugging-a-robot-program>
- Threading....

Java

- Pros
 - Acceptable speed - depending on the task it can be nearly as fast as compiled C++
 - Broad support - Numerous resources available for learning Java on the net.
 - Broadly taught in schools and colleges.
 - Similar in methodology to other object orientated languages - Subjectively better implementation.
 - Ability to incorporate wide range of 3rd party libraries.
 - Memory management is mostly automatic.
 - Threading
 - Git!
- Cons
 - Requires some level of programming knowledge
 - Moderate learning curve if you are new to programming in general
 - Garbage Collector - if misconfigured can cause significant logic delays.
 - Our experience is that java was not very responsive.
 - Threading....

Eclipse

- Open source IDE
 - Integrated Development Environment
- Default FRC IDE
- Has plugin infrastructure to extend functionality
- FRC has eclipse plugins for C++ and Java
- Contains debug environment
- Available on wide variety of Operating Systems
- Integrates very well with source control
- Very well documented both in General Eclipse Usage and FRC integration
- FRC documentation
 - <https://wpilib.screenstepslive.com/s/4485/m/13809/l/145002-installing-eclipse-c-java>
- Is responsible for replying code to robot
- Requires toolchain installation for compiling C++ cross compilation
- Requires Java Development Kit and Tools and configuration of Java environment

GIT

www.github.com

- Git documentation comes with KOP
- Source control is a very very good thing
- Allows for distributed code creation
- Allows for code reversion and history tracking
- Allows for code branching, tagging, release and Pull Requests
- Combined with Eclipse, allows for a developer to code on any OS of their choice.
- Requires some level of training to fully take advantage
- Best practices - subjective but very well tested
 - One official repository
 - Create development and production branches
 - Each developer should create their own branch / repository of the official development branch
 - Code updates to the official development branch should be done via Pull Requests
 - Code updates to the official production branch should be done via Pull Requests from the development branch
 - Learn how to use commit / check out / clone / pull / tags / merge / rebase / Pull Requests.