WPILib Robot Programming Cookbook

Robot programming recipes for cooks
of all levels

January 9, 2012

Robot Programming Cookbook   4

Robot Programming Cookbook

This is a guide for writing robot programs using the classes provided by WPILib library. It describes best practices as intended by the authors of the library. It is hoped this makes your programs better by showing you the intended use of the library and all its features.

For the 2012 FRC competition, the library goes beyond just operation of sensors, motors, and the driver station. Presented is a methodology for writing robot programs that we believe will simplify programming and debugging of your robots. There are a number of classes that are new for 2012 such as Commands, NetworkTables, significant SmartDashboard improvements, driver station based vision processing. Taken in aggregate, these changes support a new simpler approach to designing, and developing software for your robot.

Throughout the book , each new concept will be introduced with sample code taken directly from a real world robot. The examples are drawn from various 2011 FRC robots whose code has been reimplemented using the new library. One of the robot used in the examples is shown below.
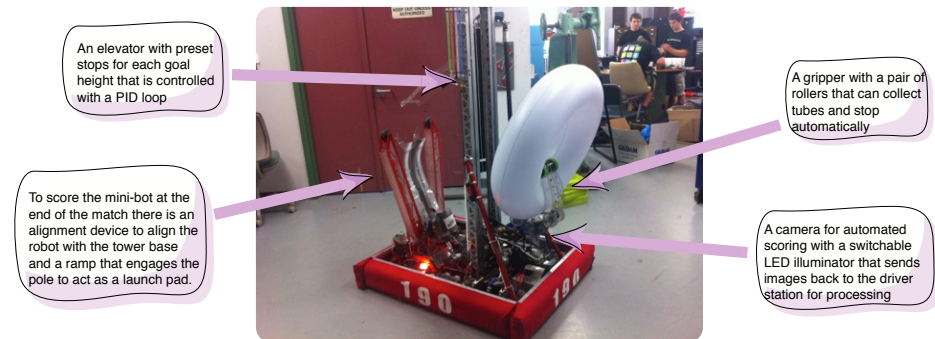
This document is organized so that you can look through it initially to get the idea about how to program robots, then later go to specific sections to solve problems that you might have.

Command-based program structure

Programs based on the WPILib library are organized around two fundamental concepts: **Subsystems** and **Commands**.

DriveLine

Wrist

MoveToScoringPosition

Gripper

StowWrist

DeployMiniBot

MiniBot
Launcher

Subsystems

Elevator

LaunchMiniBot

Commands

DriveForwardAndScore

Ball
Shooter

Arm

DriveToGoal

ElevatorToHighPosition
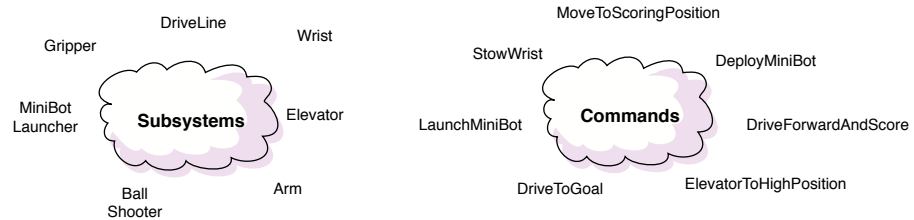
Subsystems	define the capabilities of each part of the robot and are subclasses of Subsystem.
Commands	define the operation of the robot incorporating the capabilities defined in the subsystems. Commands are subclasses of Command or CommandGroup. For convenience we suggest implementing a base class for Command called CommandBase that makes it easier to reference Subsystem objects. The default code generated by Netbeans for projects uses this technique.
	Commands run when scheduled or in response to buttons being pressed or virtual buttons from the SmartDashboard.

Commands let you break up the tasks of operating the robot into small chunks. Each command has an execute() method that does some work and an isFinished() method that tells if it is done. This happens on every update from the driver station or about every 20ms. Commands can be grouped together and executed sequentially, starting the next one in the group as the previous one finishes.

DriveToGoal → RaiseElevator → SetWristPosition → ScoreTube

Currently executing
command
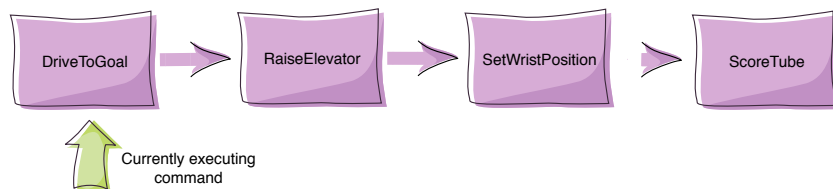
Sometimes it is desirable to have several operations happening concurrently. In the previous example you might want to set the wrist position while the elevator is moving up. In this case a command group can start a parallel command (or command group) running.

SetWristPosition

DriveToGoal

RaiseElevator

ScoreTube

At this point the elevator and the wrist are both moving at the same time

Wait for new data from the driver station

execute()

isFinished()

SetWristPosition command

execute()

isFinished()

RaiseElevator command
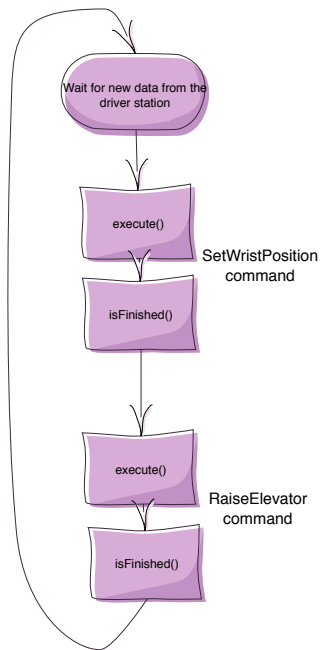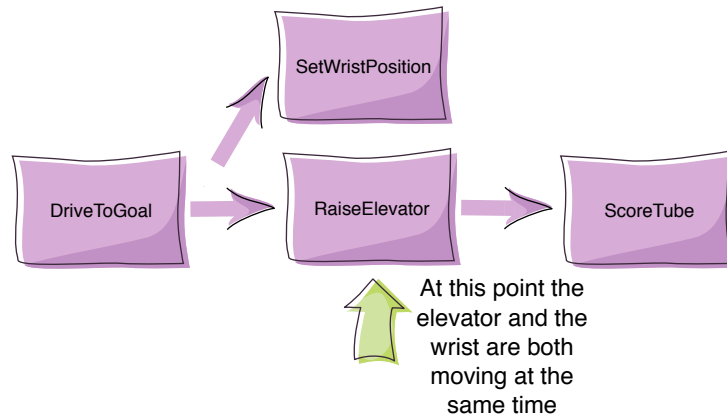
What's really happening? Each time the driver station gets new data, the periodic method of your robot template is called. It runs a scheduler that, in turn calls the execute() and isFinished() methods on each command that is currently active.

This brings up some questions about concurrency, such as ensuring that the ScoreTube command doesn't start running before *both* the SetWristPosition and RaiseElevator commands have both finished. Those will be addressed later.

It is important to understand that **creating a command or adding commands to a group doesn't start the command running**. This happens when they are explicitly scheduled using the run() method on the your command object. Usually this happens automatically in response to operator interface button presses or other events, but sometimes you have to do it manually such as starting a command for the autonomous period.

You can look at the generated main class in a command-based robot project and see the code that does this.

How to write a robot program

Using this methodology writing the robot program becomes following these steps:

1. **Create the project.**
   In Java create a CommandBasedRobotTemplate project.

2. **Create the subsystems for your robot.**
   Once you have done the initial design for the robot (you know what subsystems it has but not necessarily how they work or what sensors or motors each uses) create Subsystem classes. **This is important: you don't have to have a working or even fully designed robot for this step**. Just create the subsystem classes and methods that define the capabilities of each subsystem. As the design of each subsystem is completed, fill in the missing details in the subsystems.

3. **Create commands to make the subsystems perform the tasks to do.**
   Commands define the tasks and sequencing of tasks to perform more complex operations on the robot. Commands execute sequentially and can be tied to autonomous modes and operator interface buttons to determine when they should run.

4. **Tie the commands to the user interface buttons to get the robot operating.**
   Create an OI (operator interface) class that connects all your commands (and groups) to OI buttons and controls. Then when buttons are pressed the commands will be scheduled.

5. **Write any additional commands needed for autonomous operation.** Autonomous programs are usually multistep, more complex sequences of commands to do some task. Once you have defined the basic commands (previous step) it's easy to combine them into larger groups for doing complex operations.

6. **Instrument the robot using the SmartDashboard.** You can easily create a dashboard program for your robot that shows status as it operates. You can even have it tell you what commands are operating at any time, the status of each subsystem, or even control it through buttons and other UI elements. These features are all built into the library and the dashboard.

That's it, you're done! Go practice driving the robot before shipping it.

Note: *These steps outlined above will work to get you started although they don't necessarily have to be followed in order, and you can approach the problem in any order. This just presents an order that will work and that will be followed in the subsequent examples.*

C++ vs Java versions

The features described here are documented in both the C++ and Java WPILib libraries. There are minor differences between the two implementations and most of them are described below. The ultimate source is the reference documentation that is supplied with each language distribution. In the case of C++ there is a Microsoft Windows HTML help file and a directory containing doxygen generated documentation. For Java, the java doc comments are best.

Item	C++	Java
Getting the SmartDashboard reference	the SmartDashboard is a singleton and you should call SmartDashboard::GetInstance()->MethodName()	The SmartDashboard methods are static and you can call SmartDashboard.methodName()
Method names	Methods all start with an upper case letter and then are mixed lower and upper case	Methods all start with a lower case letter and then are mixed case
Development tools	Windriver Workbench (plug-ins coming soon)	Netbeans with plug-ins to support adding of commands and subsystems to programs

Tutorial - creating a command-based robot project in Java

This section will walk you through creating a simple project that is implemented using the command-based approach described in this document.

The command-based approach really makes a difference when writing more complex programs, but to get started here is how to create a simple program that just gets the robot driving with a joystick. Later we'll extend it to do more *interesting* things where the power of the command based programming model starts to be more apparent.

Step 1: Create the project

The method of creating a program varies depending on the chosen language. In Java, using Netbeans, there is a built-in project type called "CommandBasedRobot" that makes getting started a little easier. To create a project, select "New project..." from the file menu. Select "CommandBasedRobotTemplateProject".
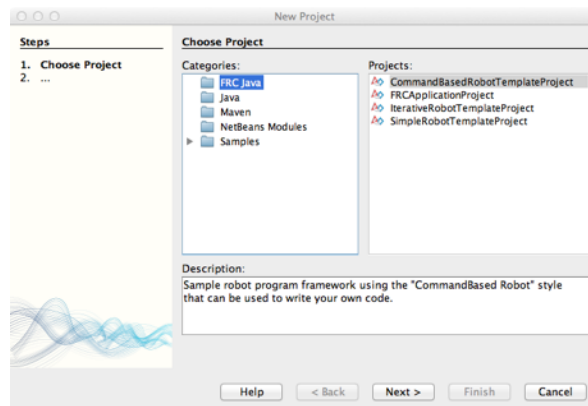
Next choose the project name, package name, and main class name. The package name is to make sure that code written by you doesn't conflict with code written by others, making it possible to share code between developers.

You can change any of these things later so don't worry too much about the names you choose.
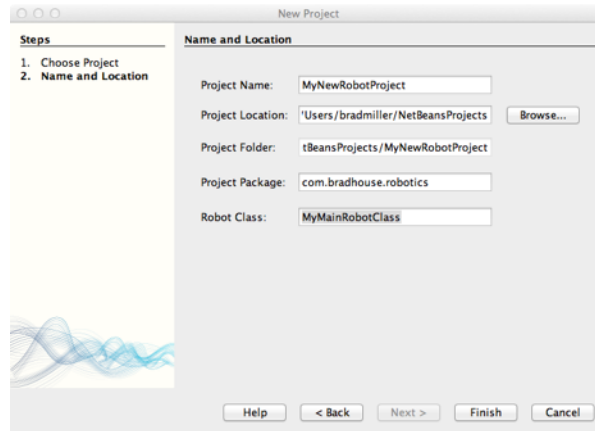
Clicking on Finish will create a project with the structure shown here. There is your package with two sub-packages, one for commands and one for subsystems. The project also includes a sample command and a sample subsystem. Those should be replaced with your own, the samples are just to show where you should put your own code. This structure is in no way required, but it might help with the organization of your own projects.
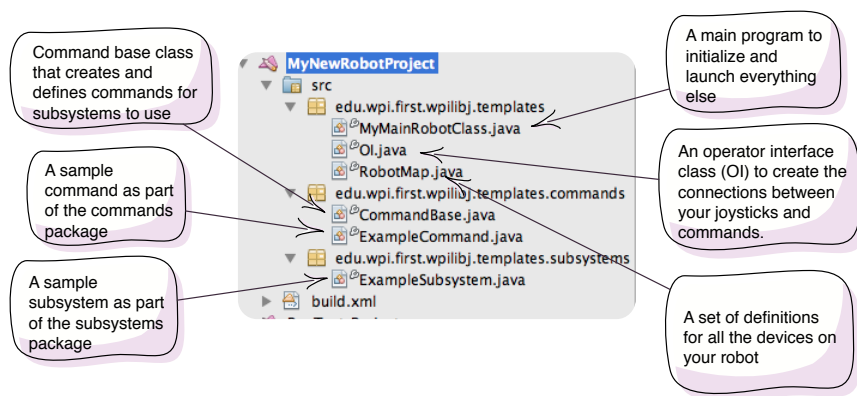
The main robot class is the starting point for your program and it has code to launch your autonomous code and guarantee that commands are scheduled at the appropriate times.

*Note: The ExampleCommand.java and ExampleSubsystem.java classes are just
         samples and can be deleted or edited into your own commands and
         subsystems.*

Step 2: Creating the robot subsystems

To create a subsystem use the "New..." menu option to create a "New Other...".
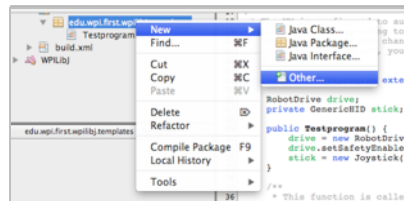
Then select Subsystem from the Command-Based Robot category.
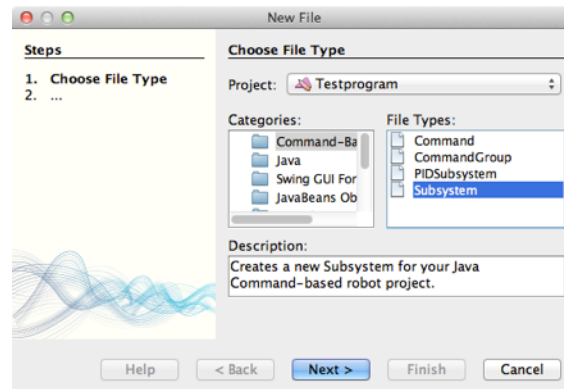
Fill in the new subsystem name and a subsystem class will be created for you.
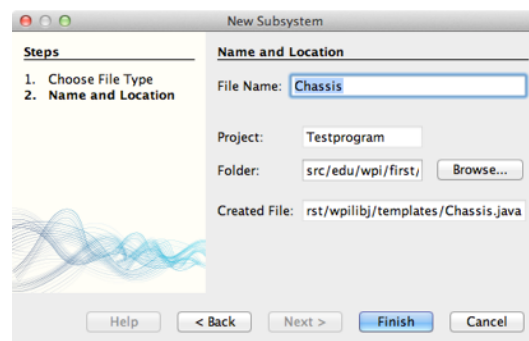
Now the new generated subsystem class can be filled out. Our simple robot only has a driveline so that is the only subsystem. The subsystem class looks like this (grey lines represent automatically generated code):

```java
public class Chassis extends Subsystem {

  RobotDrive drive;

  public void initDefaultCommand() {
    setDefaultCommand(new DriveWithJoystick());   // set default command
  }

  public Chassis() {
    drive = new RobotDrive(2, 1);
    drive.setSafetyEnabled(false);
  }

  public void straight() {   // sets the motor speeds to drive straight (no turn)
    drive.arcadeDrive(1.0, 0.0);
  }

  public void turnLeft() {    // sets the motor speeds to start a left turn
    drive.arcadeDrive(0.0, 1.0);
  }

  public void driveWithJoystick(Joystick stick) {
    drive.arcadeDrive(stick);
  }
}
```

After you create your subsystem you should add the code to instantiate the subsystem to CommandBase.java file in the CommandBase class. Only one line needs to be changed since the rest is generated automatically when the project is created. It should look roughly like this:

```java
public abstract class CommandBase extends Command {

    public static OI oi;
    public static Chassis chassis = new Chassis();

    public static void init() {
        oi = new OI();
    }

    public CommandBase(String name) {
        super(name);
    }

    public CommandBase() {
        super();
    }
}
```

Step 3: Create commands for each of the robot's operations

The robot only drives based on the joystick input so you might create a command to do exactly that called DriveWithJoystick. Since this command uses the Chassis subsystem, it is important to call the requires() method to declare that it requires exclusive use of the Chassis while running. This will ensure that any commands that we write later that also use the driveline will not conflict.

Commands can be created using the "New other..." menu option, but this time select "New Command" and fill in the class name. Once done, the command can be filled out, in this case to make the DriveWithJoystick command.

```
public class DriveWithJoystick extends CommandBase {

  public DriveWithJoystick() {
      requires(chassis);  // reserve the chassis subsystem
  }

  protected void initialize() {   // called once each time the command starts running
  }

  protected void execute() {   // called repeatedly while the command is running
      chassis.driveWithJoystick(oi.getJoystick());
  }

  protected boolean isFinished() {   // called repeatedly and determines if the
      return false;                  //    command is finished executing
  }

  // Called once after isFinished returns true
  protected void end() {   // called after the command ends for clean up
  }

  protected void interrupted() {   // called if the command is preempted or canceled
  }
}
```

Since this is the default operation that the Chassis does when it's not being asked to do anything else, in the Chassis subsystem we set TankDrive to be the default command for that subsystem.

Step 4: Edit the OI class to associate the controls with commands

In this simple example we create an instance of the Joystick object, and can return its reference, but that's all for now.

```
public class OI {
  public static final int JOYSTICK_PORT = 1;
  private Joystick stick;

  public OI() {
      stick = new Joystick(JOYSTICK_PORT);
  }

  public Joystick getJoystick() {
      return stick;
  }
}
```

Later we'll add more complex commands to get the sample robot to do more operations tied to the operator interface and those associations will go here.

Edit the main class for your robot program

The default main program is actually more than we need since there is no autonomous code in this example. Here is the sample CommandBasedRobot class.

```
public class RobotTemplate extends IterativeRobot {   // uses IterativeRobot as a base

    public void robotInit() {   // called on robot start (powered on)
        CommandBase.init();   // initialize commands and the OI (created by Netbeans)
    }

    public void teleopPeriodic() {   // called for each driver station update (every 20ms)
        Scheduler.getInstance().run();
    }
}
```

That's all you need to get started. The DriveWithJoystick command is the default command for the Chassis so the robot will drive using two joysticks during the tele-op period. It's fairly complex for such a simple program, but as you'll see, it is very easy to extend to get much more interesting behavior.

Extending the program to drive in a square pattern

Now for the interesting part of this exercise. We'll extend the program from the previous section to drive in a square pattern during the autonomous period **and** assist the driver by driving in a square pattern at the touch of the trigger button on the joystick. *You can imagine that if you were to write code to score a tube or some other complex operation in a game, it might be useful to have this code available both to the autonomous and tele-op parts of the program.*

Our sample program will use simple timing to get the robot to drive the correct distances for the sides of the square and the 90 degree turns. In a more robust program you would probably want to use sensors like encoders and gyros to get more consistent performance from your robot.

Add two commands, one to drive the robot forward for a specified period of time and another to execute a 90 degree right turn for a specified period of time. In both cases, these commands start the robot moving, wait the required time, then stop the robot. As said before, to be more accurate and repeatable, both of these commands should use feedback from sensors, but more on that later. For both of these commands it's important to require the Chassis subsystem. This will cause the default DriveWithJoysticks command to stop executing when these commands are invoked since they also require the Chassis subsystem.

```java
public class DriveStraight extends Command {
    private double m_timeout;

    public DriveStraight(double timeout) {
        m_timeout = timeout;
        requires(chassis);
    }

    protected void initialize() {
        setTimeout(m_timeout);
    }

    protected void execute() {
        chassis.straight();
    }

    protected boolean isFinished() {
        return isTimedOut();
    }

    protected void end() {
    }

    protected void interrupted() {
    }
}
```

Notice that a timeout is passed to the DriveStraight class in the constructor. When the object is scheduled (when it actually runs, not when it's created), the scheduler calls the commands initialize() method which starts the timer. This happens in the initialize() method that is called whenever the DriveStraight command is scheduled.

```
public class Turn extends CommandBase {
  private double m_timeout;

  public Turn(double timeout) {
    m_timeout = timeout;
    requires(chassis);
  }

  protected void initialize() {
    setTimeout(m_timeout);
  }

  protected void execute() {
    chassis.turnLeft();
  }

  protected boolean isFinished() {
    return isTimedOut();
  }

  protected void end() {
  }

  protected void interrupted() {
  }
}
```

The Turn command uses a timeout to time the turn in exactly the same way as the DriveStraight command in the previous section.

Now create a CommandGroup that consists of the commands required to drive in a square pattern. This group will be scheduled when a button is pressed or during the autonomous part of the match.

Remember that the arguments in the constructors for DriveStraight and Turn are the length of time to drive seconds.

```
public class DriveInASquare extends CommandGroup {

   public DriveInASquare() {
      add(new DriveStraight(1));
      add(new Turn(1));
      add(new DriveStraight(1));
      add(new Turn(1));
      add(new DriveStraight(1));
      add(new Turn(1));
      add(new DriveStraight(1));
   }
}
```

Now modify the OI class to associate the DriveInSquare command group with the trigger button on the right joystick. Whenever the joystick is pressed, the robot will drive in a square pattern. When the command is completed, the default command for the DriveLine subsystem will run - DriveViaJoysticks.

```
public class OI {
   public static final int JOYSTICK_PORT = 1;

   private Joystick stick;
   private JoystickButton trigger;

   public OI() {
      stick = new Joystick(JOYSTICK_PORT);
      trigger = new JoystickButton(stick, Joystick.ButtonType.kTop.value);
      trigger.whenPressed(new DriveInASquare());
   }

   public Joystick getJoystick() {
      return stick;
   }
}
```

And, lastly, make the robot drive in that same square pattern during autonomous by scheduling the DriveInSquare command group at the start of the autonomous() method.

```
public class RobotTemplate extends IterativeRobot {

  Command autonomousCommand;

   public void robotInit() {
     autonomousCommand = new DriveInASquare();
     CommandBase.init();
  }

  public void autonomousInit() {
     autonomousCommand.start();
  }

  public void autonomousPeriodic() {
     Scheduler.getInstance().run();
  }

  public void teleopInit() {
      autonomousCommand.cancel();
  }

  public void teleopPeriodic() {
     Scheduler.getInstance().run();
  }
}
```

That's all you need to do to drive in a square pattern on either the start of the autonomous period or the press of the right trigger joystick button. You can imagine implementing commands that help much more complex robots do tasks that are required for the yearly game challenge, but the method is always the same.

Tutorial - creating a command-based robot project in C++

This section will walk you through creating a simple project that is implemented using the command-based approach described in this document.

Installing the Workbench plugin

There is a plugin for Workbench that will make it easy to generate subsystems and commands. To install it in Workbench first make sure that you have the "Advanced Device Development" perspective selected. Then select "Help", then "Install New Software...". Fill in the update site and check the box as shown below. Then click "Next" and take all the defaults until Workbench asks to restart. Restart it and the plugin will be installed.
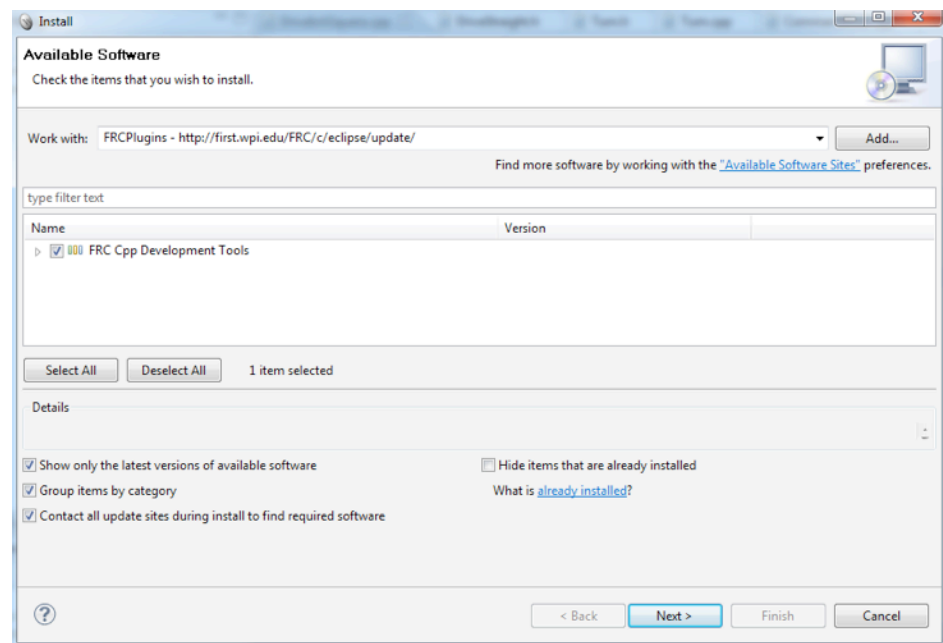
Once the plugin is installed, you can create Subsystems and Commands by right-clicking on the project name in the "Project Explorer" window and selecting "New", then type type of class to be created as shown here:
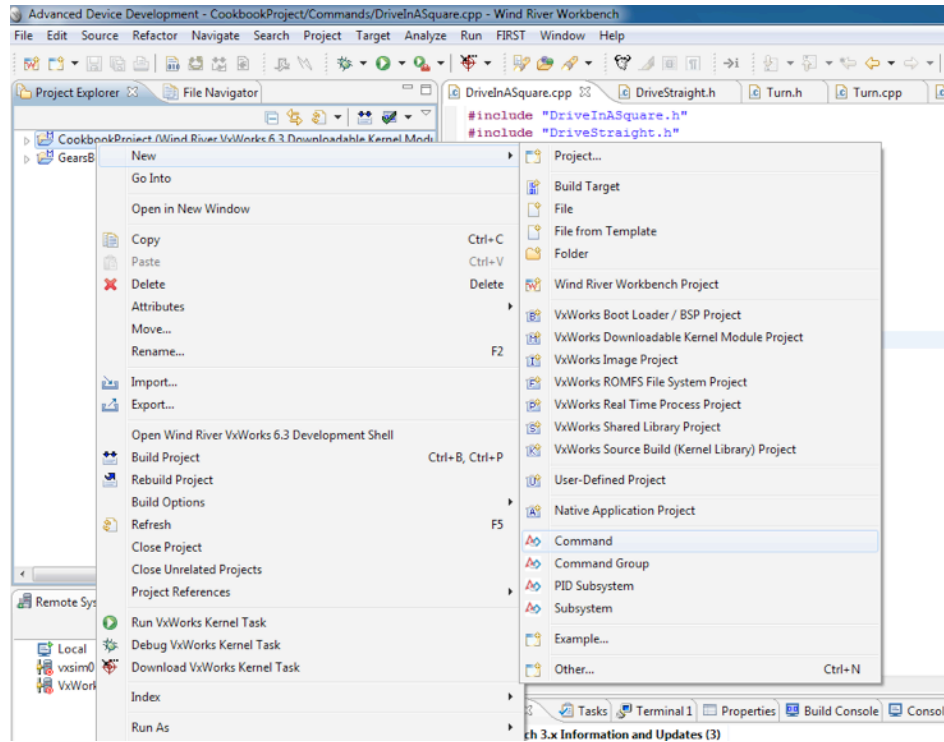
Command based approach

The command-based approach really makes a difference when writing more complex programs, but to get started here is how to create a simple program that just gets the robot driving with a joystick. Later we'll extend it to do more *interesting* things where the power of the command based programming model starts to be more apparent.

Step 1: Create the project

C++ development is done using WindRiver Workbench. To create a command-based robot project in C++ you actually create a Sample project, then add your own files to it.

Note: C++ has plugins available to create the command-based robot class files once the project is created. We are hoping to add the additional feature of creating complete projects using the plugin very soon after the 2012 kickoff. Keep an eye on the the FIRSTForge WPILib project for news about this update.

To create a sample command-based project, click "File" then "New" then "Example...".
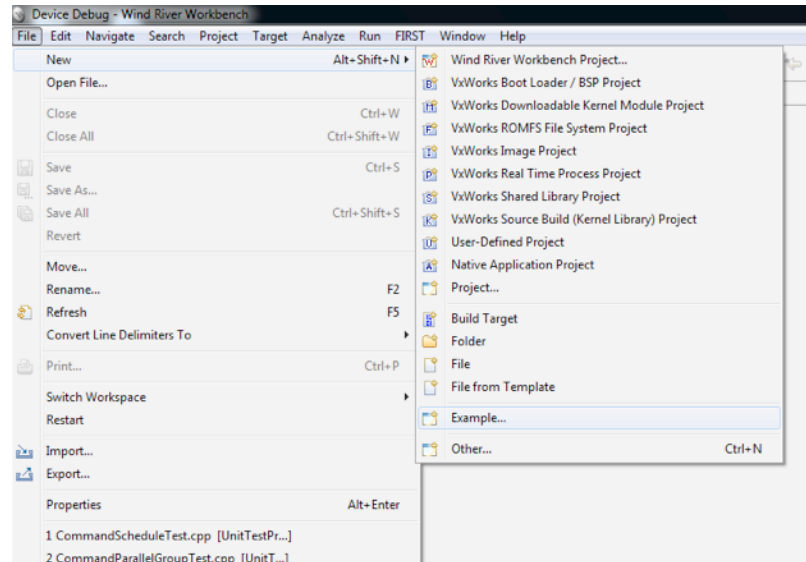
From there select a "VXWorks Downloadable Kernel Module Sample Project". Then choose Command Based Robot Template.
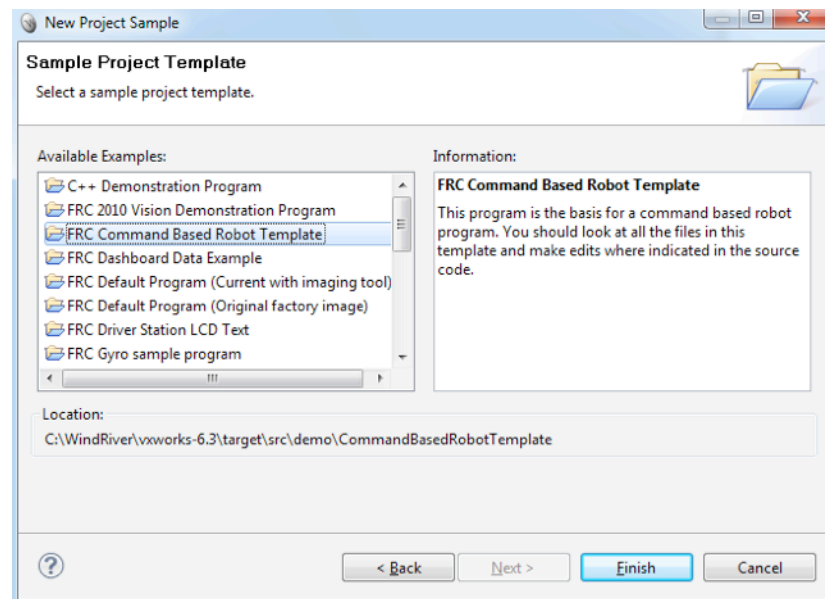
You'll get a project created in your workspace with the following structure:
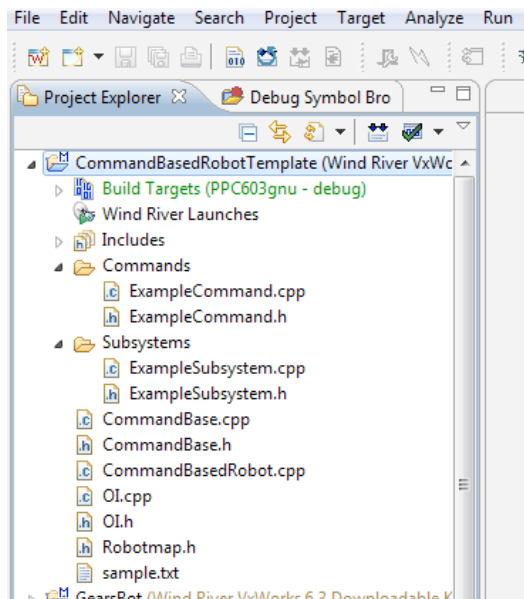
Note: *that there is a sample subsystem and command created as part of the project. These are just to show you how you can create your own subsystems and commands. These files as they are should not be part of your final project.*

Step 2: Creating the robot subsystems

To create a subsystem right-click on the project name in the "Project explorer" and select "New" then "Subsystem".
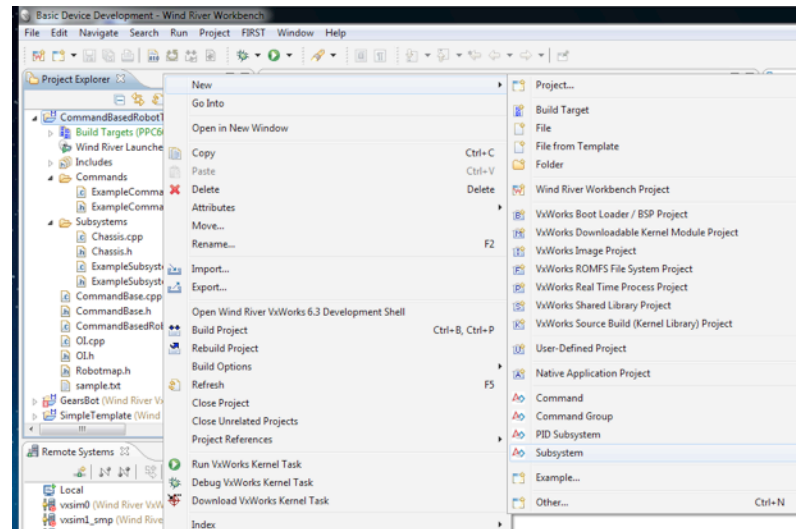
Note: You must right-click on the project name and not some other place in the project area.

Fill in the new subsystem name and a subsystem class will be created for you.
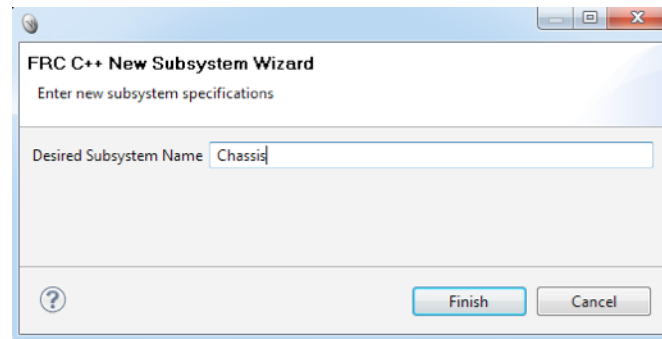
Now the new generated subsystem class can be completed. Our simple robot only has a driveline so that is the only subsystem. The subsystem class (Chassis.h and Chassis.cpp) looks like this:

```cpp
#ifndef CHASSIS_H
#define CHASSIS_H
#include "Commands/Subsystem.h"
#include "WPILib.h"

class Chassis: public Subsystem {
private:
    RobotDrive *drive;
public:
    Chassis();
    void InitDefaultCommand();
    void goStraight();
    void turnLeft();
    void tankDrive(double left, double right);
    void driveWithJoystick(Joystick *stick);
};

#endif
```

```cpp
#include "Chassis.h"
#include "../Robotmap.h"
#include "../Commands/DriveWithJoystick.h"

Chassis::Chassis() : Subsystem("Chassis") {
    drive = new RobotDrive(2, 1);
    drive->SetSafetyEnabled(false);
}

void Chassis::InitDefaultCommand() {
    SetDefaultCommand(new DriveWithJoystick());
}

void Chassis::goStraight() {
    drive->ArcadeDrive(1.0, 0.0);
}

void Chassis::turnLeft() {
    drive->ArcadeDrive(0.0, 1.0);
}

void Chassis::tankDrive(double left, double right) {
    drive->TankDrive(left, right);
}

void Chassis::driveWithJoystick(Joystick *stick) {
    drive->ArcadeDrive(stick);
}
```

After you create the Chassis subsystem you should add the code to instantiate it in CommandBase.cpp. Only one line needs to be changed since the rest is generated automatically when the project is created. It should look roughly like this:

```cpp
#include "CommandBase.h"
#include "Commands/Scheduler.h"

CommandBase::CommandBase(const char *name) : Command(name) {
}

CommandBase::CommandBase() : Command() {
}

Chassis * CommandBase::chassis = NULL;
OI* CommandBase::oi = NULL;

void CommandBase::init() {
    chassis = new Chassis();

    oi = new OI();
}
```

Step 3: Create commands for each of the robot's operations

The robot only drives based on the joystick input so you might create a command to do exactly that called DriveWithJoystick. Since this command uses the Chassis subsystem, it is important to call the requires() method to declare that it requires exclusive use of the Chassis while running. This will ensure that any commands that we write later that also use the driveline will not conflict.

Commands can be created using the "New other..." menu option, but this time select "New Command" and fill in the class name. Once done, the command can be filled out, in this case to make the DriveWithJoystick command. Here are the DriveWithJoystick.h and DriveWithJoystick.cpp files.

```
#ifndef DRIVE_WITH_JOYSTICK_H
#define DRIVE_WITH_JOYSTICK_H

#include "../CommandBase.h"

class DriveWithJoystick : public CommandBase {
public:
     DriveWithJoystick();
     void Initialize();
     void Execute();
     bool IsFinished();
     void End();
     void Interrupted();
};

#endif
```

```
#include "DriveWithJoystick.h"

DriveWithJoystick::DriveWithJoystick() {
     Requires(chassis);
}

void DriveWithJoystick::Initialize() {
}

void DriveWithJoystick::Execute() {
     chassis->driveWithJoystick(oi->getJoystick());
}

bool DriveWithJoystick::IsFinished() {
     return false;
}

void DriveWithJoystick::End() {
}

void DriveWithJoystick::Interrupted() {
}
```

Since this is the default operation that the Chassis does when it's not being asked to do anything else, in the Chassis subsystem we set TankDrive to be the default command for that subsystem.

Step 4: Edit the OI class to associate the controls with commands

In this simple example we create an instance of the Joystick object, and can return its reference, but that's all for now.

```cpp
#ifndef OI_H
#define OI_H

#include "WPILib.h"

class OI {
private:
    static const int JOYSTICK_PORT = 1;

    Joystick *stick;

public:
    OI();
    Joystick *getJoystick();
};

#endif
```

```cpp
#include "OI.h"
#include "Commands/DriveInASquare.h"

OI::OI() {
    stick = new Joystick(JOYSTICK_PORT);
}

Joystick * OI::getJoystick() {
    return stick;
}
```

Later we'll add more complex commands to get the sample robot to do more operations tied to the operator interface and those associations will go here.

Edit the main class for your robot program

The default main program is actually more than we need since there is no autonomous code in this example. Here is the sample CommandBasedRobot class.

```
#include "WPILib.h"
#include "Commands/Command.h"
#include "CommandBase.h"
#include "Commands/DriveInASquare.h"

class CommandBasedRobot : public IterativeRobot {
private:

    virtual void RobotInit() {
        CommandBase::init();
    }

    virtual void AutonomousInit() {
    }

    virtual void AutonomousPeriodic() {
    }

    virtual void TeleopInit() {
    }

    virtual void TeleopPeriodic() {
        Scheduler::GetInstance()->Run();
    }
};
```

That's all you need to get started. The DriveWithJoystick command is the default command for the Chassis so the robot will drive using two joysticks during the tele-op period. It's fairly complex for such a simple program, but as you'll see, it is very easy to extend to get much more interesting behavior.

Extending the program to drive in a square pattern

Now for the interesting part of this exercise. We'll extend the program from the previous section to drive in a square pattern during the autonomous period **and** assist the driver by driving in a square pattern at the touch of the trigger button on the joystick. *You can imagine that if you were to write code to score a tube or some other complex operation in a game, it might be useful to have this code available both to the autonomous and tele-op parts of the program.*

Our sample program will use simple timing to get the robot to drive the correct distances for the sides of the square and the 90 degree turns. In a more robust program you would probably want to use sensors like encoders and gyros to get more consistent performance from your robot.

Add two commands, one to drive the robot forward for a specified period of time and another to execute a 90 degree right turn for a specified period of time. In both cases, these commands start the robot moving, wait the required time, then stop the robot. As said before, to be more accurate and repeatable, both of these commands should use feedback from sensors, but more on that later. For both of these commands it's important to require the Chassis subsystem. This will cause the default DriveWithJoysticks command to stop executing when these commands are invoked since they also require the Chassis subsystem.

```cpp
#ifndef DRIVE_STRAIGHT_H
#define DRIVE_STRAIGHT_H

#include "../CommandBase.h"

class DriveStraight: public CommandBase {
private:
    double m_timeout;
public:
    DriveStraight(double timeout);
    void Initialize();
    void Execute();
    bool IsFinished();
    void End();
    void Interrupted();
};

#endif
```

```cpp
#include "DriveStraight.h"

DriveStraight::DriveStraight(double timeout) {
    m_timeout = timeout;
    Requires(chassis);
}

void DriveStraight::Initialize() {
    SetTimeout( m_timeout);
}

void DriveStraight::Execute() {
    chassis->goStraight();
}

bool DriveStraight::IsFinished() {
    return IsTimedOut();
}

void DriveStraight::End() {
}

void DriveStraight::Interrupted() {
}
```

Notice that a timeout is passed to the DriveStraight class in the constructor. When the object is scheduled (when it actually runs, not when it's created), the scheduler calls the commands initialize() method which starts the timer. This happens in the initialize() method that is called whenever the DriveStraight command is scheduled.

```cpp
#ifndef TURN_H
#define TURN_H

#include "../CommandBase.h"

class Turn : public CommandBase {
    private:
    double m_timeout;
public:
    Turn(double timeout);
    void Initialize();
    void Execute();
    bool IsFinished();
    void End();
    void Interrupted();
};

#endif
```

```cpp
#include "Turn.h"

Turn::Turn(double timeout) {
    m_timeout = timeout;
    Requires(chassis);
}

void Turn::Initialize() {
    SetTimeout(m_timeout);
}

void Turn::Execute() {
    chassis->turnLeft();
}

bool Turn::IsFinished() {
    return IsTimedOut();
}

void Turn::End() {
}

void Turn::Interrupted() {
}
```

The Turn command uses a timeout to time the turn in exactly the same way as the DriveStraight command in the previous section.

Now create a CommandGroup that consists of the commands required to drive in a square pattern. This group will be scheduled when a button is pressed or during the autonomous part of the match.

Remember that the arguments in the constructors for DriveStraight and Turn are the length of time to drive seconds.

```cpp
#ifndef DRIVE_IN_A_SQUARE_H
#define DRIVE_IN_A_SQUARE_H

#include "Commands/CommandGroup.h"

class DriveInASquare: public CommandGroup {
public:
    DriveInASquare();
};

#endif
```

```cpp
#include "DriveInASquare.h"
#include "DriveStraight.h"
#include "Turn.h"

DriveInASquare::DriveInASquare() {
    AddSequential(new DriveStraight(1));
    AddSequential(new Turn(1));
    AddSequential(new DriveStraight(1));
    AddSequential(new Turn(1));
    AddSequential(new DriveStraight(1));
    AddSequential(new Turn(1));
    AddSequential(new DriveStraight(1));
}
```

Now modify the OI class to associate the DriveInSquare command group with the trigger button on the right joystick. Whenever the joystick is pressed, the robot will drive in a square pattern. When the command is completed, the default command for the DriveLine subsystem will run - DriveViaJoysticks.

```cpp
#ifndef OI_H
#define OI_H

#include "WPILib.h"

class OI {
private:
	static const int JOYSTICK_PORT = 1;

	Joystick *stick;
	JoystickButton *trigger;

public:
	OI();
	Joystick *getJoystick();
};

#endif
```

```cpp
#include "OI.h"
#include "Commands/DriveInASquare.h"

OI::OI() {
	stick = new Joystick(JOYSTICK_PORT);
	trigger = new JoystickButton(stick, Joystick::kTopButton);
	trigger->WhenPressed(new DriveInASquare());
}

Joystick * OI::getJoystick() {
	return stick;
}
```

And, lastly, make the robot drive in that same square pattern during autonomous by scheduling the DriveInSquare command group at the start of the autonomous() method.

```cpp
#include "WPILib.h"
#include "Commands/Command.h"
#include "CommandBase.h"
#include "Commands/DriveInASquare.h"

class CommandBasedRobot : public IterativeRobot {
private:
    Command *autonomousCommand;

    virtual void RobotInit() {
        CommandBase::init();
        autonomousCommand = new DriveInASquare();
    }

    virtual void AutonomousInit() {
        autonomousCommand->Start();
    }

    virtual void AutonomousPeriodic() {
        Scheduler::GetInstance()->Run();
    }

    virtual void TeleopInit() {
        autonomousCommand->Cancel();
    }

    virtual void TeleopPeriodic() {
        Scheduler::GetInstance()->Run();
    }
};
```

That's all you need to do to drive in a square pattern on either the start of the autonomous period or the press of the right trigger joystick button. You can imagine implementing commands that help much more complex robots do tasks that are required for the yearly game challenge, but the method is always the same.

Note: *You might not have realized how easy it is to use some of our autonomous code during the teleop part of the competition. But with Commands, such as the DriveInASquare command, it can be triggered to run either during the autonomous period or on the press of a button by only adding the trigger to the joystick!*

Subsystems

Subsystems are classes that you write to define how each piece of the robot works. Each subsystem class is a subclass of the WPILib Subsystem class. You write methods to define each of the capabilities of the subsystem. For example, a gripper might have methods grab() and release() that grab scoring objects and release them. It is important to note that these methods would just set setpoints on the gripper, not do a timed operation. If something runs over a period of time, it should be a command. All of the state for the subsystem is contained within the subsystem, and the only public methods are the ones that describe the subsystem's capabilities.

Note: It's important that there are no waits or delays in the subsystem methods.

For subsystems like elevators there might be methods like moveToMiddle() or moveToTop() to move the elevator to the middle or top scoring position. In these cases the subsystem would use a PIDSubsystem object to move the elevator in the background to the desired position. The Elevator subsystem would simply set up the PID in its constructor, then the method moveToTop() would set a target position for the PID loop. The PID loop would then run on its own driving the elevator the appropriate position.

Basic subsystem format

A subsystem defines a piece of the robot. So inside the subsystem there should be several sections:

- The definitions of the actual devices that make up the subsystem, for example, motors, sensors, etc. Each of these object references should be private to the subsystem and never directly referenced from outside the subsystem class.

- The public methods for accessing the subsystem are the only way the rest of the robot program operates that subsystem. This ensures that if you want to change the subsystem later, you only need to look at the public methods (interface) and you're guaranteed that none of the rest of the robot code references anything else.

Best practices for writing subsystem classes

There are a number of well understood object oriented programming practices that have been developed. For the most flexibility in your programming following these practices will make code easier to debug and extend.

1. Write the Subsystem class so that it extends the WPILib Subsystem base class.

2. Make everything private unless it absolutely needs to be shared with the other elements of the program. Often there are a number of methods that are part of a class to help it implement its operations. These methods should almost always be declared private and therefore not visible to the rest of the program. This will make it possible to modify a subsystem without affecting the rest of the program.

3. Have only loose coupling between your subsystems. Coupling is the degree to which one subsystem (class) relies on the other subsystems. If a particular subsystem relies on the behavior of another subsystem then modifications to one subsystem ripple into changes in other subsystems. This makes the program much more complex to extend or change.

4. Keep the methods simple. A subsystem is in charge of getting the robot to do whatever the commands tell it to do; subsystems do not make decisions for themselves. A subsystem should not have a method where it will read from the joysticks and set its own wheel values. Instead, it should provide a method that a command can use to set the wheels (and then the command can read from the joysticks).

Simple subsystem

The claw of our sample robot is a good example of a simple subsystem. The claw has two sets of rollers that operate independently allowing the robot to pick up tubes, eject them, or roll the tubes forward or backward by only changing the motor speeds. In addition there is a limit switch connected to a digital input that is used to detect if a tube has been loaded into the claw.

The claw subsystem class is very straightforward and looks like this:

```
public class Claw extends Subsystem {

    private SpeedController top = new Victor(7);     // define some motors and a
    private SpeedController bottom = new Victor(6);  //   DigitalInput for a limit
    private DigitalInput hasTubeSwitch = new DigitalInput(3);

    public void set(double upperSpeed, double lowerSpeed) {
        top.set(-upperSpeed);
        bottom.set(lowerSpeed);
    }

    public boolean hasTube() {
        return ! hasTubeSwitch.get();   // limit switch is false when closed (grounded)
    }

    protected void initDefaultCommand() {  // no default command for claw
    }
}
```

The two capabilities of this subsystem are the set() method and the isHasTube() method. Those two methods are the only way that the rest of the program can operate the claw. And - this is important - **the class could have been written and the methods stubbed out without knowing anything about how the claw operates**. Once done, you can continue with writing the robot program.

Creating only one instance of Subsystems

Generally there would only be a single instance of any subsystem since there is only one on the robot. For example, one elevator, one drive base, one gripper. To ensure that only one instance is created, Subsystem derived objects are created by the CommandBase class. This class is the base for all Command objects and looks like this:

```
public abstract class CommandBase extends Command {
    public static OI oi;
    public static DriveTrain drivetrain = new DriveTrain();
    public static Elevator elevator = new Elevator();
    public static Wrist wrist = new Wrist();
    public static Claw claw = new Claw();

    public static void init() {
        oi = new OI();
    }

    public CommandBase(String name) { super(name); }
    public CommandBase() { super(); }
}
```

Since each subsystem is created in this class as a static variable, it only happens once. Then each of the subsystems becomes available to the commands (this is the base class) so they're easy to reference. The default commands are automatically registered by WPILib when it calls the subsystems initDefaultCommand() methods.

PID subsystems

Subsystem objects operate the devices on your robot. They should do so without having waits or polling for operations being finished. If you have a mechanism like a wrist at the end of an arm that has to do a lengthy operation like move to a particular position, the subsystem object should initiate the process, then return.

It turns out that the PID classes in WPILib solve this problem. You can start an operation by setting, for example, the target position for the wrist, then the PID object runs in the background (a separate thread) driving the motors until the wrist reaches its target. In this case the wrist has a motor to operate it and a potentiometer to determine the wrist angle.

To create the subsystem that uses a PID object a convenience class exists in WPILib called a PIDSubsystem. This has a PIDController built in and methods to simplify the code. To create a PIDSubsystem make your subsystem class extend PIDSubsystem and set the P, I, and D constants in the constructor.

```
public class Wrist extends PIDSubsystem {

    public static final double UPPER_BOUND = 0.024;
    public static final double LOWER_BOUND = UPPER_BOUND + 0.92;
    public static final double UP_POSITION = UPPER_BOUND + 0.150;

    private final Jaguar motor = new Jaguar(2);
    private final AnalogChannel pot = new AnalogChannel(4);

    public Wrist() {
        super(2.3, 0, 0);

        setSetpointRange(LOWER_BOUND, UPPER_BOUND);
        setSetpoint(UP_POSITION);

        enable();
    }
```

The setRange() method sets the minimum and maximum permissible target values for the wrist. The setSetpoint() method sets the current set-point to the up position to initialize the wrist. enable() causes the underlying PIDController to start operating the wrist as soon as the robot is enabled, bringing it to the current set-point.

There are two methods methods that you should override that will provide the input and

```
    protected double returnPIDInput() {
        return pot.getAverageVoltage() / MAX_VOLTAGE;
    }

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

output for the PID loop:

returnPIDInput()   this is the sensor that provides the feedback for the PID loop. In this case it's the fraction of the maximum voltage output from potentiometer. The imbedded PIDController uses this value to compare with the set-point to produce an error value to drive the PID loop.

usePIDOutput()   this value is the output from the PID loop that should drive (in this case) the wrist motor. It will be called from the PIDController thread to periodically update the motor speed.

There are additional functions that may be useful to override as part of the PIDSubsystem and they are documented in the reference manual for the particular language.

Finally, there are the methods for the wrist subsystem that set the position it should move to. These methods simply set the PID set-points to cause the PID loop to run the wrist in the background.

```
public void stowed() {
    setSetpoint(UP_POSITION);
}

public void tiltForScoring() {
    setSetpoint(SCORING_POSITION);
}

public void down() {
    setSetpoint(DOWN_POSITION);
}

public void jogUp() {
    setSetpointRelative(-.01);
}

public void jogDown() {
    setSetpointRelative(.01);
}
```

Notice the jogUp() and jogDown() methods move the set-point to a position relative to the current position and are controlled through a command that is attached to a momentary up/down switch.

A status method was added so that other code in the robot could determine if the wrist has reached its desired position. This was necessary for operations where the commands that operated the wrist could report being finished.

```
public boolean atSetpoint() {
    return Math.abs(getPosition() - getSetpoint()) < .1;
}
```

Commands

Commands are schedulable pieces of robot code, each one a class subclassed from Command or CommandGroup. Commands can be individual units of code or groups of commands that execute one after another. At any time there may be a number of active commands. Each time data is received from the driver station (about every 20ms) the execute() methods in all active commands are called to make more progress on that command. After executing each command the isFinished() method is called to determine if the command is done. If it is, that command no longer runs. If it was part of a command group, the next command in the group becomes active. The continues as long as there are any active commands or groups.

Command group 1

Command group 2

Command 3

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.
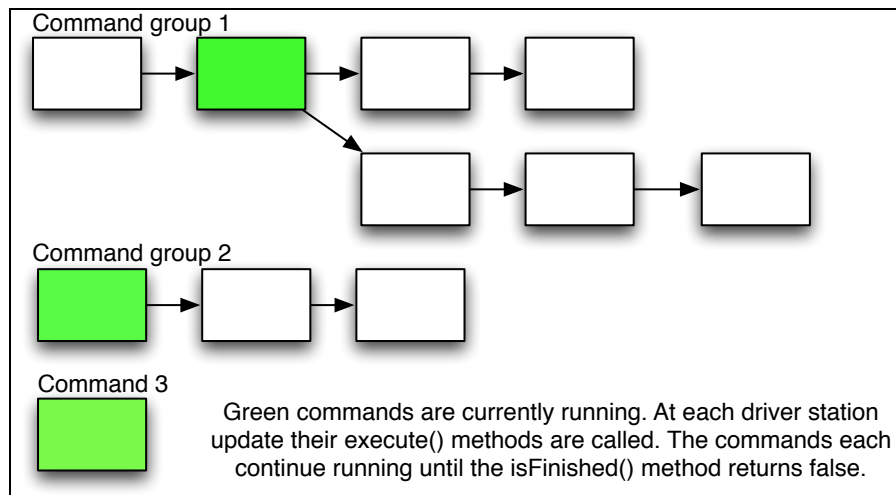
Commands and groups are associated with operator interface buttons and autonomous modes and run automatically when needed. You can write a single command that, for example, scores a tube and it can be used in the autonomous code or in the operator control code.

Basic command format

To implement a command, a number of methods are overridden from the WPILib Command class. Most of the methods are boiler plate and can often be ignored, but are there for maximum flexibility when you need it. The basic format of a command is:

```
public class MyCommandName extends CommandBase {

   public MyCommandName() {
      super("MyCommandName");
      requires(elevator);
   }

   public void initialize() {
   }

   public void execute() {
   }

   public boolean isFinished() {
      return true-if-command-is-finished;
   }
}
```

There a number of parts to this basic command class:

constructor	Might have parameters for this command such as target positions of devices. Should also set the name of the command for debugging purposes. This will be used if the status is viewed in the dashboard. And the command should require (reserve) any devices is might use.
initialize()	This method sets up the command and is called immediately before the command is executed for the first time **and every subsequent time it is started**. Any initialization code should be here.
execute()	This method is called periodically (about every 20ms) and does the work of the command. Sometimes, if there is a position a subsystem is moving to, the command might set the target position for the subsystem in initialize() and have an empty execute() method.
isFinished()	This method returns true if the command is finished. This would be the case if the command has reached its target position, run for the set time, etc.

There are other methods that might be useful to override and these will be discussed in later sections.

Simple commands

Here's an example of a command that moves the elevator to a particular position:

```java
public class ElevatorToPosition extends Command {

    double position;
    Elevator elevator = Elevator.getInstance();

    public ElevatorToPosition(double position) {
        super("MoveElevatorTo" + (int) position);

        requires(elevator);

        this.position = position;
    }

    public void initialize() {
        elevator.setPosition(position);
    }

    public void execute() {
    }

    public boolean isFinished() {
        return elevator.atSetpoint();
    }

    // the end() and interrupt() methods were removed for brevity
}
```

The constructor has a few parameters to set the position the elevator should move to. When this command is scheduled, it will run periodically.

Requiring subsystems

Suppose that you create commands to move the elevator on a robot to a particular position. Then you connect those commands to two buttons, one for the low position and one for the high position. While operating the robot, the operator presses the high position button and before it reaches the high position, the operator then presses the low position button. How is this resolved? Both commands require the use of the elevator.

Commands can indicate that they are using a particular subsystem, in this case the elevator by calling the requires() method and specifying the required subsystem. Now when the first command is moving the elevator to the high position, and the low position is pressed, the second command notices that there is already a command using the elevator and stops it (causing the interrupted() method to run in the first command) - then schedules the low position command. The result is what the operator would expect, when the second button is pressed, the elevator changes direction and starts going down.

```java
public class ElevatorToPosition extends CommandBase {

   double position;
   boolean middleColumn;
   Elevator elevator;

   public ElevatorToPosition(double position, boolean middleColumn) {
      super("MoveElevatorTo" + (int) position);
      requires(elevator);

      this.position = position;    // remember the position should move to
      this.middleColumn = middleColumn;
   }

   public void initialize() {
      elevator.setPosition(position, middleColumn);  // set the position
   }

   public void execute() {    // nothing to do here - PID loop in Elevator in control
   }

   public boolean isFinished() {
      return elevator.atSetpoint();    // check if elevator has reached the setpoint
   }

   // the end() and interrupted() methods were removed for brevity
}
```

In this example, an ElevatorToPosition command requires the elevator subsystem. When it is scheduled again at some other position, the currently running instance is stopped and the new instance is started, making the elevator head in a new direction.

Creating command groups

Often robots need to execute more complex multistep operations. **To create a single command to do all the steps you create a CommandGroup** and add each of the individual commands to it. A CommandGroup is simply a container that holds a number of commands that execute sequentially. When a command in the group finishes, the next one is started until the entire group of commands has finished.

CommandGroups can be used interchangeably with Commands. In fact, the CommandGroup is a subclass of Command. Also CommandGroups will automatically require all the subsystems of the commands in the group. There is no need to explicitly require subsystems in the command group.

Once the group is created it can be tied to a single button on your user interface - pressing a single button will cause all the commands in this group to execute sequentially.

Here is an example of a CommandGroup that collects a tube.

```
public class CollectTube extends CommandGroup {

  public CollectTube() {
    addParallel(new ElevatorToPosition(Elevator.BASE_LINE, false));
    addParallel(new WristToPosition(Wrist.DOWN_POSITION));
    addSequential(new Suck());
    addSequential(new ClawDoNothing());
    addSequential(new WristToPosition(Wrist.UP_POSITION));
  }
}
```

On our sample robot collecting a tube is a number of operations. The steps are:

1. move the elevator to the lowest position (the addParallel() method starts this running asynchronously - more on that later)

2. move the wrist down (also done asynchronously)

3. start the rollers ingesting the tube (suck - runs until a limit switch is tripped indicated the tube has been collected)

4. turn off the rollers (ClawDoNothing)

5. and move the wrist up to the scoring position.

All these are achieved by adding the individual commands to the CommandGroup called CollectTube. Each operation is a single command object.

Calling the cancel() method on a group or running a command with resource conflicts cancels (stops) the entire group. The cancel() method on the currently executing command is called (or commands in the case of parallel execution) and the group is terminated.

Commands with timeouts

Often you want a command to run for some amount of time, then stop. When commands start, the current time is automatically recorded so it becomes an easy test to see if the required time has passed. You need only to set the timeout in the initialize() method, then check it in the isFinished() method.

```java
public class MinibotLaunch extends CommandBase {
   public MinibotLaunch() {
      requires(deployment);
      requires(driveline);
      setTimeout(.5);
   }

   public void initialize() {
      deployment.arm();
      deployment.launch();
   }

   public void execute() {
      driveline.drive(-.5, -.5);
   }

   public void end() {
      deployment.endLaunch();
      driveline.drive(0, 0);
   }

   public boolean isFinished() {
      return isTimedOut();
   }
}
```

You can see that demonstrated in this sample.

In addition there is a built-in, WaitCommand command that just waits for a specified period of time at which time it ends. This is useful in a command group where you need to introduce a delay between two existing commands.

In this example, there is a 0.2 second (200 ms) delay introduced after the

```
public class Score extends CommandGroup {

   public Score() {
      addParallel(new WristToPosition(Wrist.DOWN_POSITION));
      addSequential(new WaitCommand(.2));
      addSequential(new Unsuck());
      addSequential(new WaitCommand(.5));
      addSequential(new ClawDoNothing());
   }
}
```

WristToPosition() command. This delay is used to ensure that the wrist has begun moving before starting the next command in the CommandGroup.

Commands added to a command group can have a timeout by adding it as a second parameter to either addParallel() or addSequential(). The second parameter is the length of time (in seconds) that the command should run. If the time expires on the command with the timeout parameter, then the interrupted() method on that command is called and the next command in the group is executed. This is useful in the case of a mechanism that should be fully finished operating in some amount of time, but in case the time runs out, the group should just continue.

Note: *on normal completion of a command in a group the end() method is called on the command.*

Subsystem default commands

You can specify default commands that will run for any subsystem. These default commands will run when no other command is running for the subsystem. You might want the robot to be driven with the joysticks unless there is some operator assistance command running. Suppose your robot is normally driven with the joysticks, but you have a command that will track to a target and fire when a button is pressed. You could tie the track and score command to a button on the operator interface but have the command that drives with the joysticks as a default command for the drivetrain.

Note: *The default command needs to be set outside of the constructor or it will not be set properly. To do that, add a initDefaultCommand() method and call setDefaultCommand(new <your-command>). That method will be automatically called when necessary after the constructor has run.*

Unending commands

Commands have two ways of ending: the isFinished() method returns true and the command will no longer be scheduled or another command that uses the same resources. Suppose there were two commands:

1. DriveUsingJoysticks - this command drives the robot in a pure teleoperated fashion using joysticks controlled by an operator. It never exits explicitly - its isFinished() method always returns false and is the default command for the DriveTrain subsystem.

2. ScoreOnGoal - this command uses some sensors to drive to the goal and score a game piece provided the robot is in range, but it doesn't do it completely autonomously.

```
public class DriveWithJoysticks extends CommandBase {

    public DriveWithJoysticks() {
        requires(driveLine);
    }

    protected void initialize() {
    }

    protected void execute() {
        driveLine.tankDrive();
    }

    protected boolean isFinished() {
        return false;
    }

    protected void end() {
    }

    protected void interrupted() {
    }
}
```

If the DriveUsingJoysticks command is running and a button is pressed to trigger the ScoreOnGoal command, then, provided they both require the DriveLine subsystem, the DriveUsingJoysticks command would be stopped and the ScoreOnGoal command would start running. When it finishes and returns true from its isFinished() method, then the DriveUsingJoysticks command would restart since it was the default command for the DriveLine subsystem.

Using the end() method

The end() method is called when a command completes and is a way of cleaning up or setting the state of a subsystem to a reasonable value. For example, a command might run a motor for a fixed amount of time, then have to stop it after the time runs out. Using a window motor is a good example because it can run against a stop, then stall a little longer to make sure that the motion is complete. Here's an example of a command to operate a gripper that uses the end() method to stop the motor after it's done.

```java
public class OpenGripper extends CommandBase {

   public OpenGripper() {
      requires(gripper);
      setTimeout(1.0);
   }

   protected void initialize() {
      gripper.open();
   }

   protected void execute() {
   }

   protected boolean isFinished() {
      return isTimedOut();
   }

   protected void end() {
      gripper.stop();
   }

   protected void interrupted() {
      System.out.println("Interrupted before the gripper was fully opened");
      end();
   }
}
```

This command runs for exactly 1.0 seconds, then finishes. But before the next command is scheduled, the gripper motor is stopped.

Using the interrupted() method

Sometimes a command is stopped before isFinished() returns true. This can happen if the cancel() method was called or another command with a shared requirement starts. In either of these cases, end() will not be called. Instead, interrupted() will be called.

Just like end(), interrupted() should be used to "clean up" various parts of your command. For instance, a DriveStraight command should probably stop the driveline's motors in the interrupted() method.

In many cases, your interrupted() method should do the same thing as end(). In that case have your interrupted() method call end(). Occasionally you will want to do something else as well, like this OpenGripper command which prints out that it was interrupted before stopping the motor:

```java
public class OpenGripper extends CommandBase {

  public OpenGripper() {
     requires(gripper);
     setTimeout(1.0);
  }

  protected void initialize() {
     gripper.open();
  }

  protected void execute() {
  }

  protected boolean isFinished() {
     return isTimedOut();
  }

  protected void end() {
     gripper.stop();
  }

  protected void interrupted() {
     System.out.println("Interrupted before the gripper was fully opened");
     end();
  }
}
```

Asynchronous operation - Adding parallel commands

One of the most difficult things for human operators to do is control two mechanisms at the same time. This is what computers are good at. The Scheduler has the ability to run many commands simultaneously. This happens if you have commands associated with buttons and press multiple buttons that trigger non-conflicting commands. For instance, the DriveWithJoysticks and OpenGripper command can be running simultaneously.

You can also schedule a command manually by calling the start() method on a command object. This will schedule that command immediately and any current (non-conflicting) commands will continue running.

In a CommandGroup object commands are added to the group using the addSequential() method. An alternative is to call the addParallel() method. In this case, when the group is run the parallel command will be immediately scheduled (when that command group gets to that point) as well as the next command after that.
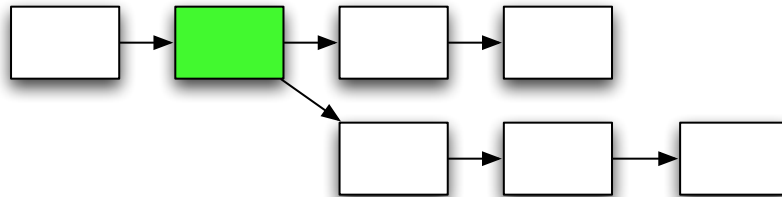
In this example the green command adds another command group as a child of the first one. Those three commands in the side-chain run asynchronously while the original group continues.

In the example below, a parallel command drives the robot for 4 seconds while the gripper is open and closed. The gripper operations run while the robot is driven.

```
public class DriveAndMoveGripper extends CommandGroup {

  public DriveAndMoveGripper() {
     addParallel(new DriveFor4Seconds());
     addSequential(new CloseGripper());
     addSequential(new OpenGripper());
  }
}
```

Notice that the command group doesn't have any requires() method calls. In CommandGroups the required subsystems for the group consist of the required subsystems of each command or group added to the group. This includes parallel commands and other groups added. This means the DriveAndMoveGripper automatically knows that it requires the driveline and gripper.

Creating uninterruptible commands

Sometimes commands should never be canceled. For instance, the example robot contains the MinibotLaunch method which had the job of lowering the ramp and launching the minibot from the 2011 game. The last thing we would want is for the driver to press a button that made the robot drive forward (that would make the minibot shoot into the middle of the field because the robot would shift out of alignment). To prevent this, MinibotLaunch marks that it requires the driveline and that it is uninterruptible.

Here is the code:

```
public class MinibotLaunch extends CommandBase {

  public MinibotLaunch() {
     requires(deployment);
     requires(driveline);
     setTimeout(.5);
     setInterruptible(false);
  }

  public void initialize() {
     deployment.arm();
     deployment.launch();
  }

  public void execute() {
     driveline.drive(-.5, -.5);
  }

  public void end() {
     deployment.endLaunch();
     driveline.drive(0, 0);
  }

  public boolean isFinished() {
     return isTimedOut();
  }

  public void interrupted() {
  }
}
```

Now another command which requires the driveline will not be able to start. In fact, calling start() on the other command will do nothing at all!

This system gets a little more interesting when an uninterruptible command is placed into a command group, as outlined in the next section.

Creating uninterruptible command groups

Because the CommandGroup extends Command, it has a built in setInterruptible() method that works the same as the one for Command. In other words, calling setInterruptible(false) in the constructor of a CommandGroup will make it uninterruptible.

The specifics get a little peculiar when a command in the group is uninterruptible while the group itself is interruptible. If the group is currently running any uninterruptible command (either in parallel or sequentially), then it will act the same as an uninterruptible command.

If the group is running an uninterruptible command in parallel and the main sequence of commands reaches a point where it requires the same Subsystem as the uninterruptible command, then it *will* cancel the uninterruptible command. The priority in a CommandGroup always goes to the main sequence of commands.

Overriding methods on command groups

CommandGroups are an extension of Commands. Because of this, they also have the initialize(), execute(), isFinished(), end(), and interrupted() methods (along with all the other ones).

You can use them to do whatever you want. For instance, you might want to reset a gyro when your autonomous starts. You can do this by writing code in the initialize() method. You might also want to end the group early, in which case you can override the isFinished() method. Be careful though, the group will only stop when isFinished() returns true. The overridden methods should call the super (parent) method to be sure that the default action continues to work.

Illegal use of commands

There are some restrictions related to the use of commands that you should be aware of. Fundamentally you can't do anything except start and stop commands once they are running. You can't do things like:

- Cancel a command that's in a CommandGroup

- Add a command to a command group after it's been started

- You can't add the same command instance to more than one command group. The problem is that the command instances store state for the command so one instance can't be running at the same time as another instance.

- You can't change requirements for a command in a group since the aggregate requirements for the group are computed when the command is added.

Generally speaking, if you do want to use the same command in multiple places in your program it's simple. Make each command a new instance by using the new operator to ensure that there is local storage for each use.

Operator interface

There are a number of features built into WPILib to help create an easy to program operator interface. You can associate commands that you write with buttons on your operator interface. The buttons can be either joystick buttons, extended I/O digital buttons, or extended I/O analog buttons. And you can easily create your own buttons that appear on the SmartDashboard.

There are two steps in connecting a command or group to a button:

1. Create an instance of the appropriate button class such as DigitalIOButton, AnalogIOButton, or JoystickButton.

2. Use the whenPressed() or whileHeld() methods on the Button object to associate the button with a Command or CommandGroup object.

When the button is pressed during the tele-op part of the game, the associated command will automatically run. In the case of whileHeld() the command will be run over and over again while the button is pressed.

You can also create your own buttons to handle special cases like modifier buttons, keyboard inputs, or any other requirements. This is described later in this document.

A good method of handling all these buttons and associations is to create a class called OI. It should have a constructor that creates the Button objects (see later) for your robot and associates commands with them.

Associating buttons with commands

Buttons are created using any of the subclasses of the Button class. These can be:

DigitalIOButton	A button connected to a digital I/O pin on the extended I/O (Cypress) module.
AnalogIOButton	A button connected with a resistor such that at 50% voltage the button changes from an off to on state.
JoystickButton	A button associated with an actual Joystick object button.
InternalButton	An internal button is a Button derived class that has a method to set when it should be considered pressed. This way your program can use anything to trigger an internal button, just call the setPressed(boolean) method.

Once button objects are created they can be associated with an instance of any command. When the button is pressed the associated command will be immediately scheduled to run. Any interruptable commands requiring the same subsystems as the new command will be stopped just before the new command starts.

```
public static final Button CLAW_COLLECT = new DigitalIOButton(8);
public static final Button CLAW_STOP_COLLECT = new DigitalIOButton(6);
public static final Button CLAW_SCORE = new DigitalIOButton(4);

CLAW_COLLECT.whenPressed(new CollectTube());
CLAW_STOP_COLLECT.whileHeld(new StopButtonPressed());
CLAW_SCORE.whenPressed(new Score());
```

You can also create your own button object types by subclassing the Button class described later.

Using analog inputs as buttons

Often when designing an operator interface for a robot there are more buttons and switches than digital I/O ports. When this happens analog ports can be used to read digital values by using a resistor to set the voltage that appears on the analog input. When the switch is open (not pressed) the voltage on the input pin is near 5.0V. When the switch is closed (pressed) the voltage becomes 0.0 volts. The AnalogIOButton class simplifies reading values that work this way.

To use it, create an instance of an AnalogIOButton with the analog input port number. A threshold value of 0.5 is built into the class. Values less than 0.5 will return true and values greater than the threshold will return false.

```
public static final Button ELEV_JOG_UP = new AnalogIOButton(5);
public static final Button ELEV_JOG_DOWN = new AnalogIOButton(7);

    ELEV_JOG_UP.whileHeld(new ElevatorJog(true));
    ELEV_JOG_DOWN.whileHeld(new ElevatorJog(false));
```

This code will run the ElevatorJog() command when either the Jog up or Jog down buttons are pressed. In the case of the sample robot, the switch is a lever that is either pushed up or down. The commands will continuously be scheduled while the switch is held in the up or down position (see next section).

Notice that the code for the analog button is almost identical to the DigitalIOButton which makes the software especially easy to use. You can put off the decision until late in the design process since the differences are contained within the button objects.

WhileHeld vs. WhenPressed methods

There are two Button class methods that can be used to cause the button value to be monitored, whenPressed() and whileHeld(). Each of these responds to button inputs in a different way.

WhenPressed   runs the associated command once when the button is pressed. After the command completes it won't be run again unless the button is released then pressed again.

WhenReleased   runs the associated command once the button is released. After the command completes it won't be run again unless the button is pressed and released again.

WhileHeld   runs the command over and over again as long as the button is held in the on position. When the button is pressed, the associated command initialize() method will be called followed by repeated calls to execute(). If the command finished while the button is still pressed, the initialize method will be called again and the cycle will repeat until the button is released. When the button is released the interrupted() method on the command will be called.

Creating custom buttons

Often a program will need to use multiple buttons or other special cases to register a button being pressed. The possibilities are endless and can't be exhaustively encoded in the library. To create your own button objects with custom triggering, simply create a class that extends Button.

The Button class has an abstract method, boolean get() that must be implemented by your Button subclass and simply returns true or false depending on the state of your button.

On our sample robot there are three lights that turn on when the camera sees the target lined up and in range. When the buttons controlling the three lights are pressed along with an elevator position, an auto-score command is triggered to drive to the goal and score a tube. This is implemented with the AndLightsButton() object.


```
public class AndLightsButton extends Button {

  Button button;

  public AndLightsButton(Button b) {
    button = b;
  }

  public boolean get() {
    return button.get()
            && OI.BLUE_LIGHT.get()
            && OI.RED_LIGHT.get()
            && OI.WHITE_LIGHT.get();
  }
}
```

Then the auto score command object is associated with the combination of the 3 lights turned on and the appropriate level button pressed at the same time. The operator

interface has two sets of three buttons that allow it to score on either set of goals (high or low).

Remember that in the game there were some sets of goals that were higher than others and the two sets of buttons selected which position to go to.

For each of the 6 positions an AndLightsButton button object is created, then associated with the command GoToTargetAutonomous(position). When the button is pressed, the correct command is scheduled.

Notice that this is exactly the same code that runs during the autonomous part of the program. The GoToTargetAutonomous command is just reused unchanged when the button is pressed to get the autonomous scoring behavior in the tele-op mode of the match.

```
    new AndLightsButton(ELEV_INNER_BOTTOM).whenPressed(new
GoToTargetAutonomous(Elevator.BOTTOM, true));
    new AndLightsButton(ELEV_INNER_MIDDLE).whenPressed(new
GoToTargetAutonomous(Elevator.MIDDLE, true));
    new AndLightsButton(ELEV_INNER_TOP).whenPressed(new
GoToTargetAutonomous(Elevator.TOP, true));
    new AndLightsButton(ELEV_OUTER_BOTTOM).whenPressed(new
GoToTargetAutonomous(Elevator.BOTTOM, false));
    new AndLightsButton(ELEV_OUTER_MIDDLE).whenPressed(new
GoToTargetAutonomous(Elevator.MIDDLE, false));
    new AndLightsButton(ELEV_OUTER_TOP).whenPressed(new
GoToTargetAutonomous(Elevator.TOP, false));
```

SmartDashboard

The SmartDashboard is a Java program that will display robot data in real time. Typically you would run the SmartDashboard on your driver station laptop although it **could also run on a different computer on the same network**. The SmartDashboard helps you with these things:

- Displays robot data of your choice while the program is running. It can be displayed as simple text fields or more elaborately in many other display types like graphs, dials, etc.

- Displays the state of the robot program such as the currently executing commands and the status of any subsystems

- Displays buttons that you can press to cause variables to be set on your robot

- Allows you to choose startup options on the dashboard that can be read by the robot program

The displayed data is automatically formatted in real-time as the data is sent from the robot, but you can change the format or the display widget types and then save the new screen layouts to be used again later.

And with all these options, it is still extremely simple to use. To display some data on the dashboard, simply call one of the SmartDashboard methods with the data and its name and the value will automatically appear on the dashboard screen.
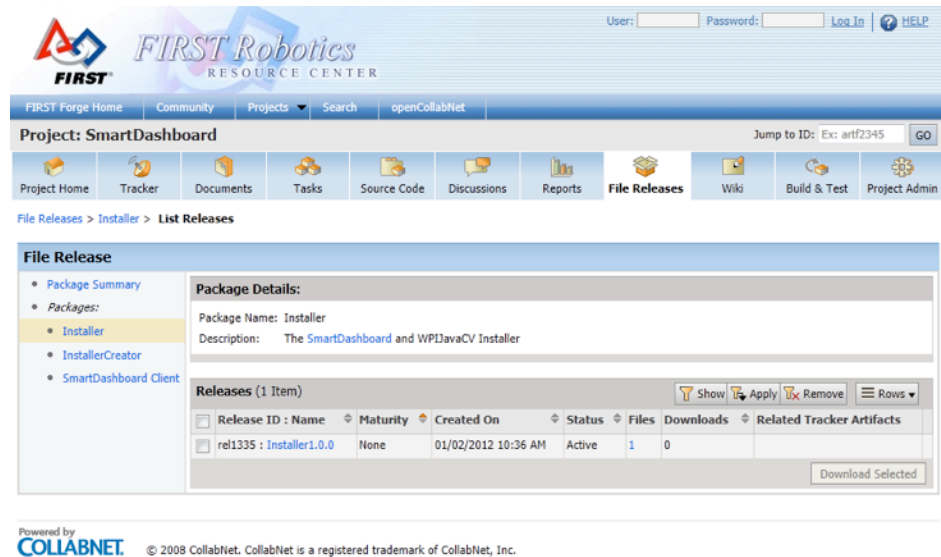
The following sections in this document list the ways of displaying data on the SmartDashboard and interacting with it. Many of these are one-way, that is the robot supplies a value and it appears on the dashboard. Others are bi-directional - the program causes the widget to be displayed, but you can interact with it from the dashboard and change values on the robot. You should skim through the document to see all the options available to you.

Installing SmartDashboard

The first step to installing the SmartDashboard is to get the installer from the webpage pictured below, which can be found at

http://firstforge.wpi.edu/sf/frs/do/listReleases/projects.smartdashboard/frs.installer

Check off the latest version and press "Download Selected"

Once you have the zip file, open it up and run the executable inside. You will then be led through the installation wizard.

Congratulations! You have installed the SmartDashboard.

If it was installed on the driverstation computer, then it should run automatically when you log in.

If it was installed on a different computer, then the SmartDashboard can be run by double-clicking the SmartDashboard.jar file in C:\\Program Files\SmartDashboard

Note: This will only work for Windows. To install this on a mac is unsupported; however, if you would like to have the SmartDashboard on a mac without camera support then you can visit the SmartDashboard project on firstforge.wpi.edu and read the wiki for how to do that

Viewing robot data values

Viewing subsystem status

You can view the status of any subsystem by adding it to the SmartDashboard. What you'll see is command that is currently running that requires that subsystem. At any time there might not be any commands using a subsystem, and in that case nothing will be displayed.

```
SmartDashboard.putData(gripper);
```

This will display the gripper status on the SmartDashboard while the robot program is running and is the basis for an excellent way of helping to debug the code.

Gripper ---

In this case the Gripper status shows no commands currently running on it. If a command is scheduled for the Gripper subsystem it will be shown instead of the dashes.

Viewing currently running commands

Robot programs are just sets of running commands, many of which can run concurrently. To better understand what your program is doing at any time, you can use the SmartDashboard to view which commands are running. The Scheduler is a class in WPILib that maintains the list of all the running commands and advances them as they complete. To view which commands are running at any time, add the following line of code to the robotInit() method in your program:

```
SmartDashboard.putData("SchedulerData", Scheduler.getInstance());
```

it will cause a list of all the commands currently running to be displayed on the SmartDashboard. For debugging purposes you will also have the option of canceling any of the running commands from the dashboard interface.

Displaying the camera video feed

You can display the camera video feed on the SmartDashboard.

Testing commands

PID tuning with the SmartDashboard

There are several options for viewing and updating PID controllers using the SmartDashboard. In all cases, using the class instance as a parameter to the SmartDashboard putData method will display the PID object.

SendablePIDController	This is a subclass of PIDController and can be used without any of the command based programming classes. It will display using the editable format shown below.
PIDSubsystem	Displaying the PIDSubsystem will show the status of the subsystem like any others, that is the name of the command currently manipulating the subsystem will be displayed. Putting the SmartDashboard into edit mode (View/Editable) and right-clicking on the widget and changing it to PIDEditor will display the PID tuner shown below.
PIDCommand	Displaying the PIDCommand will show status of the command like any others, that is the command will display it's name and button to either start of cancel it. Putting the SmartDashboard into edit mode (View/Editable) and right-clicking on the widget and changing it to PIDEditor will display the PID tuner shown below.

You can tune your PID controller by editing the values in the text fields in the widget. The values will be stored locally in the classes in the running code. You must record the values that work correctly and add them to your code so it will work without the need of the tuner widget.

Creating buttons that run commands

Using a SendableChooser for selecting autonomous programs

Often a team will develop multiple autonomous programs or have other startup options that need to be set before the match starts. Typically this is done using various switches, buttons, and other operator interface hardware. These methods usually require complex programming of driver station inputs and sometimes obscure command sequences.

Using the SmartDashboard you can easily program in those types of options. The SendableChooser class will let you list a number of named options on the dashboard that can be selected using the mouse. Then the robot program can retrieve the options at any time and run the appropriate code.

Here is an example of using the SendableChooser to pick one of two autonomous commands to run at the start of the autonomous period in the game. In this program fragment there are two command classes called DoSomeAutonomousStuff and DoSomeOtherStuff that represent possible autonomous options.

The steps to create the choice field for autonomous options are:

1. Create a SendableChooser object

2. Add each of the autonomous commands to it. The one added using addDefault() will be selected when the widget is created on the screen.

3. Add the SendableChooser to the SmartDashboard by using SmartDashboard.putData(). This will create a set of radio buttons (chooser) to let the operator select which option should be used.

4. In the autonomous method, get the selected mode and start it.

```
public class RobotTemplate extends CommandBasedRobot {

   Command autonomousCommand;
   SendableChooser autoChooser;

   public void robotInit() {
      NetworkTable.initialize();

      autoChooser = new SendableChooser();
      autoChooser.addDefault("Default program", new DoSomeAutonomousStuff());
      autoChooser.addObject("Experimental auto", new DoSomeOtherStuff(););
      SmartDashboard.putData("Autonomous mode chooser", autoChooser);
   }

   public void autonomous() {
      autonomousCommand = (Command) autoChooser.getSelected();
      autonomousCommand.start();
   }
}
```
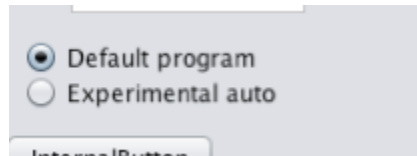
On the screen in the SmartDashboard you will see the two options displayed with radio buttons and the operator can choose the one to use:

Notice that the option labeled "Default program" is checked by default because it was added to the SendableChooser using the addDefault() method.

Displaying and editing robot preferences

Often it is useful to store values on the robot and read them when your program starts up. These are things like PID values, ports, or other preferences that you might use. There is a preference class in WPILib that will store those values in a file on the cRIO and allow you to read them while your program is running. The preferences can be viewed and modified using the SmartDashboard preferences widget. To display it, select "View/Add/Robot Preferences".

Note: To view robot preferences on the SmartDashboard you must have a Preferences object instantiated on the robot.

Creating custom widgets

You can create your own widgets (SmartDashboard display elements) by writing simple Java classes that extend StaticWidget. The full procedure is described in the Wiki at: http://firstforge.wpi.edu/sf/wiki/do/viewPage/projects.smartdashboard/wiki/HomePage.

Once you have created your own widgets, it can be customized using the properties windows just like the built-in widgets.

How it works

The SmartDashboard displays named data and in particular display data of type SmartDashboardNamedData or SmartDashboardData. You will see that many of the classes in WPILib are designed to display their status on the dashboard and are subclasses of SmartDashboardData or SmartDashboardNamedData. Any of those objects can just be passed to SmartDashboard.putData("name", some-SmartDashboardData-value) or to SmartDashboard.putData(some-SmartDashboardNamedData-value).

Preferences class

Often in programs you might like to store data that is persistent between program runs. For example, potentiometer calibration values that measure the height of an arm are dependent on the exact mounting of the potentiometer. The preference class allows you to store data in the flash memory on the micro-controller with names for each value. Later you can retrieve the values by asking for them by name.

Preference values can also be viewed using the SmartDashboard. See the description in the SmartDashboard section of this document.

NetworkTables for communications

NetworkTables allows the programmer to easily send data between a laptop (either the driver station laptop or another laptop) and the robot. The data transfer is:

• Bidirectional - the data can be send from either the driver station to the robot or from the robot to the driver station.

• Named - the data that is sent is named, that is your program on either side can set a value for a name and the value becomes available on the other side of the connection.

• Full notification - you can be notified when data arrives through a callback mechanism so the program doesn't have to poll for new values.

• Multiple named tables - the data is stored in a tabular format where each row has a name and value. The tables are themselves named, and you can have multiple tables.

• Transactional model - you can ensure that a group of values are updated simultaneously. For example, camera target information for a given image should all be updated together. You can force a set of values to not be made available until it is all there.

Laptop based Vision system

Vision-based program often requires a large amount of computation, sometimes more than can be comfortably run on the robot controller (cRIO). In this case, an alternative implementation is to run the vision code on the driver station computer or another laptop that is on the network with the driver station computer. The camera can be connected directly to a switch on the robot and the robot controller never has to process images, and can devote all of its resources to responding to the other sensors and driving the robot.

Typically camera images are processed in real-time and reduced to some small set of values, for example, the distance to a target, x position of the target, size of the target, etc. WPILib has implemented a set of classes that easily facilitate the passing of data between the robot and the driver station - and this communication is bi-directional. These classes are collectively called NetworkTable and allow the programmer to set the values of named data on one one side and the data is automatically updated on the other side.

To do the image processing there are a number of libraries available. National Instruments provides the NI Vision library and we have also provided Java wrappers for the open source OpenCV library. The OpenCV library will be discussed in the following sections of this document.

The Java wrapped OpenCV library (WPIJavaCV) can be run independently of any other program; however, the tutorials in this section will be designing additions to the SmartDashboard, so that your image processing can run right on the SmartDashboard.

Installing the Java vision components

The Java vision components will be installed automatically when you run the SmartDashboard installer.

Look at the "Installing SmartDashboard" section to see how to get the installer.

Structure of a simple vision program

Before we start programming, we need to create a new java project. For these tutorials, we will use Netbeans, but any other IDE will work.

Open up Netbeans and create a new java project (go to the "Java" folder and select "Java Application"). You may want to uncheck the "Create Main Class" option; you will not need one for this project. Call the project whatever you like.

Find out what jars are needed or write this into Netbeans

Once you have the project, right click it and select "Properties". Go to the "Libraries" section in the properties window.............

Now we are ready to begin coding. Create a new class and make it extend WPICameraExtension. The WPICameraExtension class is designed to do most of the work interfacing with the camera and the SmartDashboard for you. What we need to do is edit the processImage(...) method. So you should have a java file which looks like this:

```
public class BasicCameraExtension extends WPICameraExtension {

  @Override
  public WPIImage processImage(WPIColorImage rawImage) {
    return super.processImage(rawImage);
  }
}
```

This processImage(...) method will be called repeatedly by the SmartDashboard. The input is the image we get from the camera, and the image we return is what we want drawn on the SmartDashboard (which is usually the camera image with some outlines or shapes drawn on it).

For this simple vision program, we will just show the red values of the image. This is just the following code:

```
public class BasicCameraExtension extends WPICameraExtension {

  @Override
  public WPIImage processImage(WPIColorImage rawImage) {
    return rawImage.getRedChannel();
  }
}
```

This project is now ready to be stuck onto the SmartDashboard. The first step is to build the project. Once its built, go to the folder that you put this project and open up the "dist" folder. Copy the .jar file inside the folder and move it into "C:\\Program Files\SmartDashboard\extensions" Now run the SmartDashboard by double-clicking "C:\\Program Files\SmartDashboard\SmartDashboard.jar".

Once SmartDashboard is running, you can see your code in action by selecting "View>Add...>*your class name*" from the file menu.

Sending data to the robot

Detecting circular targets

Measuring distance to a target

Using Smart Dashboard for debugging

The SmartDashboard can be a very useful debugging tool for understand what your robot program is doing in real time. Some of the useful features (described in detail in other parts of the book) are:

- Showing values from sensors, motors, or intermediate calculations
- Looking at the state of your robot program, what commands are running, what commands are using subsystems, communications state, etc.
- Looking at the robot camera output to observe what it is viewing
- Running or stopping individual commands to test them without having to write test programs. Adding a command to the SmartDashboard will display as a Cancel button while the command is running and a Start button when the command is not running.

Glossary

Button

Command

JoystickButton

NetworkTable

PID

PIDCommand

PIDSubsystem

SmartDashboard

Subsystem