Using Strongback

strongback

Published with GitBook

Table of Contents

Strongback is an open source Java library that makes it easier for you to write and test your robot code for FIRST Robotics Competition. You use it along with WPILib for Java, and you deploy it and your codebase to your RoboRIO. Strongback is:

- **Simple** - Strongback's API is simple and natural, and it uses Java 8 lambdas and fluent APIs extensively to keep your code simple and readable.
- **Safe** - Strongback itself uses WPILib for Java, so there are no surprises and behavior remains consistent.
- **Testable** - When your code uses Strongback, you can test much more of your robot code on your computer without requiring real robot hardware.
- **Timely** - Strongback's commands and asynchronous functions share a single dedicated thread. This dramatically reduces context switches in the JVM, and on the dual-core RoboRIO results in consistent and reliable periods required for control system logic.

This book gives an introduction to Strongback and explains the benefits of using it, and it explains how to get started with Strongback by downloading and adding it to your robot codebase. The book then goes into detail about the different parts of Strongback and how you can use them. Finally, since Strongback is open source, it explains how you can contribute to the Strongback codebase, website, and even this book.

Background

FIRST Robotics Competition, or FRC, is an international high school robotics competition. Every year since 1992, FIRST kicks off the FRC season by announcing a new multi-team game for robots. For the next six weeks, teams of high school students and adult mentors work to design, build, program, and test large game-playing robots. Each robot must satisfy all of the safety and game requirements, are often 6 feet tall and weigh up to 120 pounds, and perform in autonomous and tele-operated modes. Teams compete with and against other teams in their region, and the top teams advance to the FIRST World Championships.

In 2015, FRC introduced a new control system for all teams to use. The new system is easier to use, and is more capable than its predecessors. At its center is the new main robotics controller called the RoboRIO. Each team programs their robot using one of three languages: C++, Java, and National Instruments LabVIEW.

Those teams writing their robot code in Java or C++ use a library called WPILib to interface with the hardware. The WPILib makes it relatively easy to work with a variety of sensors, actuators, motor controllers, compressors, pneumatic solenoids, and other devices. Both libraries offer nearly identical functionality and a similar API. This API changed slightly in 2015, but for the most part has been stable for several years.

Using the WPILib can still be challenging, though. Since the library requires hardware, testing your robot code often requires running your code on the robot. That means that most tests are manual and time consuming. It also means that, during competition season, an FRC team's programmers can do little testing before the team's robot is ready.

Designing testable robot code is critically important, yet doing this with WPILib requires a lot of extra work. This is where Strongback comes in.

Introducing Strongback

Strongback is an open source Java library that makes it easier for you to write and test your robot code for FIRST Robotics Competition. You use it along with WPILib for Java, and you deploy it and your codebase to your RoboRIO. Strongback is:

- **Simple** - Strongback's API is simple and natural, and it uses Java 8 lambdas and fluent APIs extensively to keep your code simple and readable.
- **Safe** - Strongback itself uses WPILib for Java, so there are no surprises and behavior remains consistent.

- **Testable** - When your code uses Strongback, you can test much more of your robot code on your computer without requiring real robot hardware.
- **Timely** - Strongback's commands and asynchronous function to share a single dedicated thread. This reduces or eliminates context switches in the JVM, and on the dual-core RoboRIO results in consistent timing required for control system logic.

There are several major parts of Strongback, and you can choose which parts you want to use: hardware components, command framework, data and event recorders, asynchronous execution framework, and simple logging. We'll cover each of these areas in a separate chapter.

Simple Examples

Before we get too far, let’s take a look at a few simple examples. The goal of this chapter is to give you a quick peek into what robot code that uses Strongback looks like. Just remember that you’ll probably see some code here that you might not understand, but we’ll definitely cover all of it in later chapters.

Really simple tank drive

Here’s the compmlete code for a really simple tank-drive robot, with one motor on each side driven by a simple arcade-style joystick:

```
package org.omgrobots.example;

import org.strongback.components.Motor;
import org.strongback.components.ui.ContinuousRange;
import org.strongback.components.ui.FlightStick;
import org.strongback.drive.TankDrive;
import org.strongback.hardware.Hardware;

public class SimpleTankDriveRobot extends edu.wpi.first.wpilibj.IterativeRobot {

    private static final int JOYSTICK_PORT = 1; // in driver station
    private static final int LEFT_MOTOR_PORT = 1;
    private static final int RIGHT_MOTOR_PORT = 2;

    private TankDrive drive;
    private ContinuousRange driveSpeed;
    private ContinuousRange turnSpeed;

    @Override
    public void robotInit() {
        // Set up the robot hardware ...
        Motor left = Hardware.Motors.talon(LEFT_MOTOR_PORT);
        Motor right = Hardware.Motors.talon(RIGHT_MOTOR_PORT).invert();
        drive = new TankDrive(left, right);

        // Set up the human input device ...
        FlightStick joystick = Hardware.HumanInterfaceDevices.logitechAttack3D(JOYSTICK_P
        driveSpeed = joystick.getPitch();
        turnSpeed = joystick.getRoll().invert();
    }

    @Override
    public void teleopInit() {
    }

    @Override
    public void teleopPeriodic() {
        drive.arcade(driveSpeed.read(), turnSpeed.read());
    }

    @Override
    public void disabledInit() {
    }
}
```

As you can see, there's not much logic there. The `robotInit()` method does most of the work, and it really is just setting up some hardware components. The `teleopPeriodic()` method is called repeatedly by the `IterativeRobot` base class, and it just reads the drive and turn speeds and sends them to the `TankDrive` object.

This robot only uses Strongback's hardare components, and doesn't use any of the other Strongback features.

Tip	Motors vs motor controllers Robot code really doesn't care about what kind of *motors* are used on the robot. Instead, the code needs to know what kind of *motor controllers* are used. In our code above, we use the `Hardware.Motors.talon(…)` method to create each "motor", although on our robot hardare that really corresponds to a motor and motor controller pair.

Adding more motors

What if our robot had *two* motors on each side, but was otherwise exactly the same. We'd only have to change our `robotInit()` method to create a few extra motors:

```
package org.omgrobots.example;

import org.strongback.components.Motor;
import org.strongback.components.ui.ContinuousRange;
import org.strongback.components.ui.FlightStick;
import org.strongback.drive.TankDrive;
import org.strongback.hardware.Hardware;

public class SimpleTankDriveRobot extends edu.wpi.first.wpilibj.IterativeRobot {

    private static final int JOYSTICK_PORT = 1; // in driver station
    private static final int LF_MOTOR_PORT = 1;
    private static final int LR_MOTOR_PORT = 2;
    private static final int RF_MOTOR_PORT = 3;
    private static final int RR_MOTOR_PORT = 4;

    private TankDrive drive;
    private ContinuousRange driveSpeed;
    private ContinuousRange turnSpeed;

    @Override
    public void robotInit() {
        // Set up the robot hardware ...
        Motor left = Motor.compose(Hardware.Motors.talon(LF_MOTOR_PORT),
                                   Hardware.Motors.talon(LR_MOTOR_PORT));
        Motor right = Motor.compose(Hardware.Motors.talon(RF_MOTOR_PORT),
                                    Hardware.Motors.talon(RR_MOTOR_PORT)).invert();
        drive = new TankDrive(left, right);

        // Set up the human input device ...
        FlightStick joystick = Hardware.HumanInterfaceDevices.logitechAttack3D(JOYSTICK_P
        driveSpeed = joystick.getPitch();
        turnSpeed = joystick.getRoll().invert();
    }

    @Override
    public void teleopInit() {
    }

    @Override
    public void teleopPeriodic() {
        drive.arcade(driveSpeed.read(), turnSpeed.read());
    }

    @Override
    public void disabledInit() {
    }
}
```

Here, we're creating two motors on the left and two on the right (with each motor using a Talon motor controller), but we're using the `Motor.compose(Motor,Motor)` method to create a single `Motor` object that wraps two other `Motor` objects. After all, we're always going to control the two left motors exactly the same way, so we never need to address them individually. We then pass those two composite `Motor` objects into the `TankDrive` constructor.

Everything else is the same!

Tank drive robot with autonomous mode

What if we want to add some autonomous behavior to our robot? To keep things simple, we want our robot to drive forward at 50% speed for 5 seconds. First, we want to configure Strongback in the `robotInit()` method:

```
@Override
public void robotInit() {
    // Set up Strongback using its configurator. This is entirely optional, but we're
    // events or data so it's better if we turn them off. All other defaults are fine
    Strongback.configure().recordNoEvents().recordNoData().initialize();

    ...
}
```

We then want to override the `IterativeRobot.autonomousInit()` method, so we'll add this:

```
@Override
public void autonomousInit() {
    // Start Strongback functions ...
    Strongback.start();
    // Submit a command ...
    Strongback.submit(new Command(5.0) {
        @Override
        public boolean execute() {
            drive.tank(0.5, 0.5);
            return false;
        }
        @Override
        public void end() {
            drive.stop();
        }
    });
}
```

This `Strongback.submit(…)` method takes a `Command`, which we instantiate as an anonymous class. The resulting command runs for 5 seconds, and each time the command runs the `execute()` method will drive forward at 50% power. After 5 seconds, the command's `end()` method is automatically called and it stops the drive motors.

Tip	Command classes Although you can create `Command` subclasses using anonymous classes, it's usually better to do this only for really simple logic. You will probably make most of your `Command` classes regular classes, since they're much easier to test and reuse in both autonomous and teleop modes.

Finally, we have to add some logic to `disabledInit()` to stop the drive motors (in case the robot is disabled before the 5 second drive time has elapsed) and also shutdown Strongback:

```
    @Override
    public void disabledInit() {
        drive.stop();
        // Tell Strongback that the robot is disabled so it can flush and kill commands.
        Strongback.disable();
    }
```

Here's the complete class:

```
package org.omgrobots.example;

import org.strongback.Strongback;
import org.strongback.command.Command;
import org.strongback.components.Motor;
import org.strongback.components.ui.ContinuousRange;
import org.strongback.components.ui.FlightStick;
import org.strongback.drive.TankDrive;
import org.strongback.hardware.Hardware;

public class SimpleTankDriveRobot extends edu.wpi.first.wpilibj.IterativeRobot {

    private static final int JOYSTICK_PORT = 1; // in driver station
    private static final int LF_MOTOR_PORT = 1;
    private static final int LR_MOTOR_PORT = 2;
    private static final int RF_MOTOR_PORT = 3;
    private static final int RR_MOTOR_PORT = 4;

    private TankDrive drive;
    private ContinuousRange driveSpeed;
    private ContinuousRange turnSpeed;

    @Override
    public void robotInit() {
        // Set up Strongback using its configurator. This is entirely optional, but we're
        // events or data so it's better if we turn them off. All other defaults are fine
        Strongback.configure().recordNoEvents().recordNoData().initialize();

        // Set up the robot hardware ...
        Motor left = Motor.compose(Hardware.Motors.talon(LF_MOTOR_PORT),
                                   Hardware.Motors.talon(LR_MOTOR_PORT));
        Motor right = Motor.compose(Hardware.Motors.talon(RF_MOTOR_PORT),
                                    Hardware.Motors.talon(RR_MOTOR_PORT)).invert();
        drive = new TankDrive(left, right);

        // Set up the human input device ...
        FlightStick joystick = Hardware.HumanInterfaceDevices.logitechAttack3D(JOYSTICK_P
        driveSpeed = joystick.getPitch();
        turnSpeed = joystick.getRoll().invert();
    }

    @Override
    public void autonomousInit() {
        // Start Strongback functions ...
        Strongback.start();
        Strongback.submit(new Command(5.0) {
            @Override
            public boolean execute() {
                drive.tank(0.5, 0.5);
                return false;
            }
            @Override
            public void end() {
                drive.stop();
            }
        });
    }

    @Override
    public void teleopInit() {
    }

    @Override
    public void teleopPeriodic() {
        drive.arcade(driveSpeed.read(), turnSpeed.read());
    }

    @Override
    public void disabledInit() {
        drive.stop();
        // Tell Strongback that the robot is disabled so it can flush and kill commands.
        Strongback.disable();
    }
}
```

Add data logging

One other cool feature of Strongback is its data logging system, which records one or more channels of data that you specify. Stronback uses a single thread that runs commands, data and event logging, button monitoring, etc., and that thread does all of this every 20 milliseconds (although you can control the period via Strongback's configuration).

Let's add data logging to our robot. First, we want to change Strongback's configuration to remove the `recordNoData()` and `recordNoEvents()` methods, since we now want to record them. The first two lines of `robotInit()` now become:

```
    @Override
    public void robotInit() {
        // Set up Strongback using its configurator. All defaults are fine for this examp
        Strongback.configure().initialize();

        ...
    }
```

And, we want to tell Strongback what to record, and this is done at the *end* of the `robotInit()` method:

```
    @Override
    public void robotInit() {
        ...
        // Set up the data recorder to capture the left & right motor speeds.
        Strongback.dataRecorder()
                  .register("Left motors", left)
                  .register("Right motors", right);
    }
```

This records one channel called "Left motors" whose values will be the instantaneous speed (in percentage) sent by our code to the left motor controllers, and another channel called "Right motors" whose values will be the instantaneous speed (in percentage) sent by the right motor controllers.

Once again, the data recorder by default will make measurements every 20 milliseconds, so if we run our robot (including autonomous mode) for 2 minutes, we'll have recorded a time history with 6000 measurements for each motor! We can then disable the robot, copy the data file from the RoboRIO to our computer, and use Strongback's command line tool to convert the binary data file to a comma-separated (CSV) file:

```
$ strongback.sh log-decoder -f strongback-data-1.log -o strongback-
```

Before we go on, though, let’s look at the complete robot code for our hypothetical robot:

```
package org.omgrobots.example;

import org.strongback.Strongback;
import org.strongback.command.Command;
import org.strongback.components.Motor;
import org.strongback.components.ui.ContinuousRange;
import org.strongback.components.ui.FlightStick;
import org.strongback.drive.TankDrive;
import org.strongback.hardware.Hardware;

public class SimpleTankDriveRobot extends edu.wpi.first.wpilibj.Ite

    private static final int JOYSTICK_PORT = 1; // in driver statio
    private static final int LF_MOTOR_PORT = 1;
    private static final int LR_MOTOR_PORT = 2;
    private static final int RF_MOTOR_PORT = 3;
    private static final int RR_MOTOR_PORT = 4;

    private TankDrive drive;
    private ContinuousRange driveSpeed;
    private ContinuousRange turnSpeed;

    @Override
    public void robotInit() {
        // Set up Strongback using its configurator. All defaults a
        Strongback.configure().initialize();

        // Set up the robot hardware ...
        Motor left = Motor.compose(Hardware.Motors.talon(LF_MOTOR_P
                                   Hardware.Motors.talon(LR_MOTOR_P
        Motor right = Motor.compose(Hardware.Motors.talon(RF_MOTOR_
                                    Hardware.Motors.talon(RR_MOTOR_
        drive = new TankDrive(left, right);

        // Set up the human input device ...
        FlightStick joystick = Hardware.HumanInterfaceDevices.logit
        driveSpeed = joystick.getPitch();
        turnSpeed = joystick.getRoll().invert();

        // Set up the data recorder to capture the left & right mot
        Strongback.dataRecorder()
                  .register("Left motors", left)
                  .register("Right motors", right);
    }

    @Override
    public void autonomousInit() {
        // Start Strongback functions ...
        Strongback.start();
        Strongback.submit(new Command(5.0) {
            @Override
            public boolean execute() {
                drive.tank(0.5, 0.5);
                return false;
            }
            @Override
            public void end() {
                drive.stop();
            }
        });
    }

    @Override
    public void teleopInit() {
    }

    @Override
    public void teleopPeriodic() {
        drive.arcade(driveSpeed.read(), turnSpeed.read());
    }

    @Override
    public void disabledInit() {
        drive.stop();
        // Tell Strongback that the robot is disabled so it can flu
```

```
            Strongback.disable();
        }
    }
```

Read to get started?

Hopefully this chapter gave you a little picture of what robot code looks like when it uses Strongback. Of course, we just touched the surface, so continue on with the next chapters to get a much more detailed understanding of all of Strongback's power.

Getting Started

Strongback is designed to work with the WPILib for Java library, and to cleanly integrate with your robot Eclipse projects created with the WPILib plugin for Eclipse. Be sure to install the latest stable version of the 2016 WPILib plugins for Eclipse, or if you've installed an earlier version simply use Eclipse's "check for updates" feature to automatically upgrade the WPILib plugins. **And, if you've not yet created a new robot project using the WPILib plugins, do that now so that WPILib can fully initialize itself.**

Eclipse, Java, WPILib, and Strongback work on Windows, OS X, and Linux. The only thing different between these platforms is how to download and install Strongback.

Downloading and installing Strongback

Download the latest version of Strongback that is compatible with your version of the WPILib library. Extract the ZIP file (or compressed TAR file) to your home directory, which is the same place where the "wpilib" software is installed.

Tip	**Upgrading** If you've previously installed an older version of Strongback, we highly recommend removing the existing `strongback` directory before installing the new version. You do not need to modify your path, Eclipse workspaces, or Eclipse projects.

OS X and Linux

If you're on OS X or Linux, you can a terminal to download and install the latest version, which as of March 6, 2016 is "1.1.7":

```
$ curl -OL https://github.com/strongback/strongback-java/releases/d
$ unzip strongback-1.1.7.zip -d ~/
```

Then add Strongback's `bin` directory to your path. Again, on OS X or Linux you can do this from the command line:

```
$ export PATH=~/strongback/java/bin:$PATH
```

You'll need to do this in each terminal, since it is not permanent. You can make it permanent, however, by adding these lines to you shell configuration file (e.g., `~/.bashrc` if you use the bash shell):

```
export STRONGBACK_HOME=/Users/<yourusername>/strongback
export PATH=${STRONGBACK_HOME}/java/bin:${PATH}
```

Windows

If you use Windows, point your favorite browser to https://github.com/strongback/strongback-java/releases, download the latest ZIP file, and extract it into `C:\Users\<yourUsername>`. You should end up with a `C:\Users\<yourUsername>\strongback` directory. If you've already created a robot project in Eclipse using the WPILib plugins, you should also see an existing `C:\Users\<yourUsername>\wpilib` directory.

Then use Window Control Panel (e.g., "My Computer" > "Properties" > "Advanced" > "Environment Variables" > "Path") to append the `;C:\Users\<yourUsername>\strongback\java\bin` directory to the `PATH` system path, using a semicolon to separate it from any existing value, and of course using your correct username.

Tip	**Terminal on Windows** Throughout this book we give command that start with `$`. OS X and Linux users can simply copy and paste these as-is, but Window's `cmd.exe` command window uses a very different syntax. We trust that you can translate these commands so they'll work on Windows — for exaple, using backward slashes instead of forward slashes in paths. Also, be sure to use `strongback.bat` rather than `strongback.sh`. However, we suggest considering a better terminal. GitHub for Windows is a lighter-weight native installation of the Git tools, but more importantly it includes **Git BASH**, an excellent full-featured terminal application that iooks and behaves like terminals on OS X and Linux. In fact, all of the commands throughout this document should work as-is in Git BASH, including the use of the `strongback.sh` command.

What's in the distribution

The distribution ZIP or compressed TAR file contains the Strongback JARs, source JARs, command line tool, and JavaDoc. The extracted directory looks like this:

```
strongback
   /java
       /ant
```

```
            build.properties
            build.xml
        /bin
            strongback.bat
            strongback.sh
        /eclipse
            Strongback.userlibraries  (*generated by 'new-project')
        /javadoc
            ...
        /lib
            strongback.jar
            strongback-sources.jar
        /lib-tests
            strongback-testing.jar
            strongback-testing-sources.jar
            fest-assert-1.4.jar
            fest-assert-1.4-sources.jar
            fest-util-1.1.6.jar
            fest-util-1.1.6-sources.jar
            hamcrest-core-1.3.jar
            hamcrest-core-1.3-sources.jar
            junit-4.11.jar
            junit-4.11-sources.jar
            metrics-core-3.1.0.jar
            metrics-core-3.1.0-sources.jar
        /lib-tools
            strongback-tools.jar
            strongback-tools-sources.jar
        /templates
            build.properties.template
            build.xml.template
            classpath.template
            project.template
            Robot.java.template
            TestRobot.java.template
        README.md
        COPYRIGHT.txt
        LICENSE.txt
        strongback.properties
```

Note that Strongback does **not** contain the WPILlib libraries or Eclipse plugins. As mentioned above, you must first install those using the official downloads and installation instructions.

Checking the Strongback version

Since 1.1.0, Strongback's command line utility can display the version of Strongback that you are using. In a terminal, go to your Eclipse workspace directory (or where you want your new robot project) and run the `strongback.sh` executable on OS X, Linux, or even the Git Bash terminal on Windows:

```
$ strongback.sh version
```

If you are using the Windows command shell, you should use the `strongback.bat` batch file instead of `strongback.sh`.

```
$ strongback.bat version
```

Adding Strongback to your project

Strongback includes a command line utility to update an WPILib robot project with the necessary libraries to use Strongback. In a terminal, go to your Eclipse workspace directory (or where you want your new robot project) and run the `strongback.sh` executable on OS X, Linux, or even the Git Bash terminal on Windows:

```
$ strongback.sh new-project -e -p com.myteam.robot -r MyRobotProjec
```

Note: If you are using the Windows command shell, you'll need to use the `strongback.bat` batch file instead of `strongback.sh`. From this point on, though, all the examples in the book will refer to `strongback.sh`, and you'll just need to remember to use the Windows batch file instead.

This command creates a new robot Eclipse project (via the `-e`) named `MyRobotProject` in the `MyRobotProject` directory, stubs out an IterativeRobot subclass in the `com.myteam.robot` package, and adds a stub unit test class to the project. The project is all set to use the WPILib and Strongback libraries.

Then, import the project into your Eclipse by going to **File→Import**, chooose **General→Existing Projects into Workspace**, press 'Next', and then browse to your workspace directory and check the new project.

The `new-project` command does have a few options, and will display them when you ask it for help:

```
$ strongback.sh new-project -h
```

Strongback User Library

Strongback Eclipse projects are set up to use a "Strongback" user library in Eclipse. That user library defines the location of all of the necessary Strongback JAR files (including the source JARs) and the JavaDoc.

When you create a new project using the Strongback command line tool, the tool will automatically look for your Eclipse workspace metadata and try to define this "Strongback" user library. If it does define it, the tool will ask you to restart your Eclipse workspace so that Eclipse picks up the definition.

If you keep your `.metadata` directory located elsewhere, then the tool will not be able to automatically define the user library, and the tool will tell you to import the Strongback user library into your Eclipse workspace. To do this:

1. Open your Eclipse preferences
2. In the dialog box, navigate to **Java→Build Path→User Libraries** in the left handle panel of the preferences dialog
3. Press the "Import" button on the right side of the dialog box, and use the file chooser to locate the `Strongback.userlibraries` file inside the `java/eclipse` directory in the Strongback installation. For example, on OS X or Linux this will be `~/strongback/java/eclipse/Strongback.userlibraries`. (If you don’t see this file, run Strongback’s command line tool to create a new project; that tool will generate the file for your local installation.)
4. Press "OK" to import the user library and close the import dialog.
5. Press "OK" to save the changes to the preferences and close the dialog box.

Compiling your project

Your project should be ready to compile in Eclipse, but you can also use Ant to compile, run the unit tests, and deploy code to the robot. For example, in a terminal go to your project's directory:

```
$ cd MyRobotProject
```

You can run Ant with one or more targets:

- `clean` removes all of the generated files.
- `compile` compiles all of the project's Java source code.
- `test` runs `compile` if necessary and then runs all of the unit tests in the `testsrc` directory of the project (and will `compile` if needed).
- `deploy` runs `compile` if necessary and then deploys the JAR files to the RoboRIO.

So, to force a recompile and run all of the tests, you can run this:

```
$ ant clean test
```

To deploy the code to your robot (just like with WPILib's Ant scripts):

```
$ ant clean deploy
```

And of course, we recommend run the tests before deploying to be sure that nothing breaks:

```
$ ant clean test deploy
```

Testing with Strongback

You should add more JUnit test classes to the `testsrc` directory. When you do, you can run any or all of the tests from within Eclipse using the JUnit runner, or you can run them on the command line using Ant:

```
$ ant test
```

Design Philosophy

Strongback uses several design patterns throughout its API. This section talks abstractly about the philosophy. You'll see later on how these characteristics manifest themselves in different parts of the library.

Many of these philosophies are common practice in professional software. Together, they make code easy to read, understand, reason about, write, and test.

Functional interfaces

The Strongback API uses lots interfaces, and we tried to keep the interfaces as small and focused as possible. You'll also see quite a few default methods on those interfaces — this is something that was added in Java 8, and it allows the interface to define a method *and* a default implementation. Implementing classes get this default method for free, although they can always override any of the methods in the interfaces. Interface methods that do not have a default implementation are referred to as "abstract methods".

Functional interfaces were also new in Java 8: a *functional interface* is any interface that has exactly one abstract method. Here's an example of a functional interface used in Strongback:

```java
public interface DistanceSensor {

    /**
     * Gets the current distance, in inches, relative to the zero-point.
     *
     * @return the distance in inches
     */
    public double getDistanceInInches();

    /**
     * Gets the current value of this {@link DistanceSensor} in feet.
     *
     * @return the value of this {@link DistanceSensor}
     * @see #getDistanceInInches()
     */
    default public double getDistanceInFeet() {
        return getDistanceInInches() / 12.0;
    }
}
```

This interface represents a sensor that can detect the distance from the sensor to some object placed in front of the sensor. It has one abstract method, `getDistanceInInches()` that returns the distance in inches, and one default method `getDistanceInFeet()` that returns the distance in feet and that is implemented in terms of the abstract method. This is a functional interface in Java 8, which means it can be implemented with only a single function that takes no parameters and returns a double. You can do this the traditional way by defining a class that extends the `DistanceSensor` interface, or you can use a lambda to define that single function.

Using lambdas is incredibly powerful. Let's imagine a method that takes a `DistanceSensor` :

```
public double computeShootingAngle( DistanceSensor sensor ) {
    ...
}
```

If we have a `DistanceSensor` object, then we can pass it as a parameter to the method:

```
DistanceSensor mySensor = ...
double angle = computeShootingAngle(mySensor);
```

But what if the geometry of our robot and the field were such that we needed to always add 10 inches to our the distance measured by our sensor? We can very easily use a lambda to define a function that took no parameters and return a double:

```
DistanceSensor mySensor = ...
double angle = computeShootingAngle(()->mySensor.getDistanceInInches() + 10.0);
```

Or we might not even have a real distance sensor, but we instead calculate the distance based using vision. In that case, we probably have a method somewhere that returns the calculated distance:

```
public class VisionModel {
    ...
    public double estimateDistance() {...}
    ...
}
```

Our class doesn't have to implement or contain a `DistanceSensor` , and its `estimateDistance()` method isn't even named the same as `DistanceSensor.getDistanceInInches()` . Yet we still can have `computeShootingAngle` use that method by passing a lambda that calls our vision model:

```
VisionModel visionModel = ...
double angle = computeShootingAngle(()->visionModel.estimateDistance());
```

Or, since `estimateDistance()` takes no parameters and returns a double, we can alternatively pass a *method reference* rather than a lambda. The following code is identical to the previous snippet:

```
VisionModel visionModel = ...
double angle = computeShootingAngle(visionModel::estimateDistance);
```

Functional interfaces, lambdas, and method references take some getting used to, but they are a very powerful new addition to Java 8 and help to keep your code as small and simple as possible.

Immutable

Where possible Strongback tries to use *immutable* objects. Immutable objects appear to external observers as fixed and unchangeable. This means that you can pass around immutable objects without worrying that some other component might change them. And because immutable objects never change, concurrent access by multiple threads is trivial: no synchronization, no volatiles, no locking, and no race conditions to worry about.

Immutable classes also tend to be simpler, since there's no need for setters and fields are often final.

All immutable Strongback classes are annotated with the `@Immutable` annotation.

Injection

When an object requires references to one or more other objects, we very often will require all of those references be passed into the constructor, and the class will then store those references using `final` fields. This is a form of *injection*, which is just a fancy way of saying that when a component is created it should be handed all the objects it needs.

For example, imagine a class that requires a `Motor` and `DistanceSensor`. The constructor should take references to these objects and use them throughout its lifetime. When used on the robot, the code can pass in the real motor and distance sensor into the component, while a test case can easily pass in a mock motor and mock distance sensor.

This is also true for more significant, resource-intensive objects. Classes should never create threads; instead, they should be handed an `Executor` or even a `Supplier<Thread>` which they can use to run code asynchronously.

The bottom line is many class should depend only upon objects *injected* via the constructor, and should never *know* how to find one. This helps keep these classes easily *testable*.

Minimal requirements

When using injection, be sure to the constructor uses the most minimal interface or class. For example, if a component uses the current speed of a motor and only uses the `getSpeed()` method. So the constructor should not take a `Motor` or `LimitedMotor`, but should instead take a `SpeedSensor`.

This helps minimize dependencies, and it documents the developer’s intent (in our example, that the component should only be *reading* the speed). It also makes testing easier, since tests need to mock fewer and less complicated objects.

Fluent APIs

It’s fairly common in Java libraries for methods like setters to simply return `void`. After all, the methods change the object and you already have a reference to the object (otherwise you wouldn’t be able to invoke the method).

Where possible, Strongback methods that normally would return `void` often return a reference to the target of the method. The sole purpose is to allow methods to be chained together. And, like many modern Java libraries, we’ve abandoned the outdated pattern of naming all setters with `set*` and instead carefully name our methods to be easily readable.

An API that uses these patterns is called a *fluent API*, and if properly designed these APIs make your code *easy to write* (by leveraging code completion features in your IDE) and *easy to read*.

Consider the `AngleSensor` class, which defines these two methods (among others):

```
public interface AngleSensor extends Zeroable {

    /**
     * Gets the angular displacement in continuous degrees.
     * @return the positive or negative angular displacement
     */
    public double getAngle();

    /**
     * Change the output so that the current angle is considered to be 0.
     * @return this object to allow chaining of methods; never null
     */
    @Override
    default public AngleSensor zero() {...}

    ...
}
```

This normally `zero()` might return `void`, but having the method return the same `AngleSensor` (e.g., `this`) allows us to chain multiple methods together:

```
AngleSensor sensor = Hardware.AngleSensors.potentiometer(3,28.8).zero();
```

Another more meaningful example is that Strongback can be easily configured using a chain of method calls:

```
Strongback.configure().recordNoEvents()
                      .recordCommands()
                      .useSystemLogger(Level.DEBUG)
                      .initialize();
```

A third example is a bit more involved, but except for the imports this is a complete, working example of a tank drive robot with 4 motors:

```
public class SimpleTankDriveRobot extends IterativeRobot {

    private static final int JOYSTICK_PORT = 1; // in driver station
    private static final int LF_MOTOR_PORT = 1;
    private static final int LR_MOTOR_PORT = 2;
    private static final int RF_MOTOR_PORT = 3;
    private static final int RR_MOTOR_PORT = 4;

    private TankDrive drive;
    private ContinuousRange driveSpeed;
    private ContinuousRange turnSpeed;

    @Override
    public void robotInit() {
        // Set up the robot hardware ...
        Motor left = Motor.compose(Hardware.Motors.talon(LF_MOTOR_PORT),
                                   Hardware.Motors.talon(LR_MOTOR_PORT));
        Motor right = Motor.compose(Hardware.Motors.talon(RF_MOTOR_PORT),
                                    Hardware.Motors.talon(RR_MOTOR_PORT)).invert();
        drive = new TankDrive(left, right);

        // Set up the human input controls for teleoperated mode.
        FlightStick joystick = Hardware.HumanInterfaceDevices.logitechAttack3D(JOYSTICK_P

        // We want to use the Logitech Attack 3D's throttle as a "sensitivity"
        // input to scale the drive speed and throttle, so we'll map it
        // from it's native [-1,1] to a simple scale factor of [0,1] ...
        ContinuousRange sensitivity = joystick.getThrottle().map(t -> (t + 1.0) / 2.0);

        // scale the pitch ...
        driveSpeed = joystick.getPitch().scale(sensitivity::read);

        // scale and invert the roll ...
        turnSpeed = joystick.getRoll().scale(sensitivity::read).invert();

        // Set up the data recorder to capture the left & right motor speeds
        // (since both motors on the same side should be at the same speed,
        // we can just record the speed of the composed motors) and the sensitivity.
        // We have to do this before we start Strongback...
        Strongback.dataRecorder()
                  .register("Left motors", left)
                  .register("Right motors", right)
                  .register("Sensitivity", sensitivity.scaleAsInt(1000));
    }

    @Override
    public void teleopInit() {
        // Start Strongback functions ...
        Strongback.start();
    }

    @Override
    public void teleopPeriodic() {
        drive.arcade(driveSpeed.read(), turnSpeed.read());
    }

    @Override
    public void disabledInit() {
        // Tell Strongback that the robot is disabled so it can flush and kill commands.
        Strongback.disable();
    }
}
```

Hardware Components

Strongback provides abstract interfaces for common actuators, sensors, user controls, and other devices. This makes your subsystems simpler and more focused. And when you use these interfaces in your robot code, your robot can use hardware implementations on the real robot while your tests can use mock implementations so you can test more without the robot hardware.

The following subsections describe each of Strongback's low-level physical component abstractions:

- Accelerometers that measure acceleration in one or more axes. Hardware-based accelerometers include the ADXL345 and the RoboRIO's built-in accelerometer.
- Angle sensors that measure relative changes in angles, heading, and/or angular rates. Examples of hardware-based angle sensors are encoders, potentiometers, and gyroscopes.
- Distance sensors are sensors that that measure relative distances. Potentiometers and ultrasonic sensors are examples of hardware-based distance sensors.
- Switches are sensors that have an active state when triggered and an inactive state when not triggered. Conceptually switches are relatively abstract, but can represent multiple kinds of hardware devices, including Reed switches, limit switches, proximity sensors, magnetic switches, and user-interface buttons. Fuses are a specialization of a switch that can be reset.
- Motors are devices that can be set to operate at a controllable speed. Each Strongback's motor represents a physical motor and motor controller pair. For example, a CIM motor and Talon SRX controller would be modeled in Strongback as a single `Motor` object.
- Solenoids are physical devices that can be extended and retracted. Examples of physical solenoid devices include electromagnetic solenoids that move an armature when activated. Pneumatic cylinders and the electromagenetic solenoid valves that control the air are together modeled as one single- or double-acting solenoid in Strongback.
- Drives are components with multiple motors typically used for robot locomotion. Strongback has a *tank drive* with multiple motors on each side that are controlled together while those on opposite sides are controlled independently; tank drives can be controlled using arcade-style, tank-style (aka skid-steer), or cheesy-drive human

interfaces. It also offers a *mechanum drive* that has independently-controllable motors on each corner wheel; mechanum drives can be controlled using polor or cartesian systems, and can be used to control holonomic chassis as well.

- Miscellaneous components such as clocks, temperature sensors, voltage sensors, current sensors, and even larger physical composite devices such as the Power Distribution Panel.

Accelerometers

A hardware accelerometer measures the acceleration along one, two, or three axes. The ADXL345 is an accelerometer that was included in the 2014 Kit of Parts, and in 2015 the RoboRIO had a built-in accelerometer.

This section describes Strongback's multiple interfaces that represent physical accelerometer devices.

Accelerometer

The `org.strongback.components.Accelerometer` interface represents a single-axis accelerometer capable of sensing acceleration in one direction:

```
public interface Accelerometer {

    /**
     * Get the acceleration in g's.
     * @return the acceleration
     */
    public double getAcceleration();
}
```

As described above, an `Accelerometer` instance has a single method that returns the ratio of instantaneous acceleration to that of standard gravity at sea level. This scalar ratio is commonly know as g.

This interface measures acceleration in a single direction based upon the orientation of the particular sensor.

TwoAxisAccelerometer

Unlike the single-axis Accelerometer interface, the `org.strongback.components.TwoAccessAccelerometer` interface provides accelerations along two axes:

```java
public interface TwoAxisAccelerometer {

    /**
     * Get the X-axis accelerometer.
     *
     * @return the accelerometer for the X-axis; never null
     */
    public Accelerometer getXDirection();

    /**
     * Get the Y-axis accelerometer.
     *
     * @return the accelerometer for the Y-axis; never null
     */
    public Accelerometer getYDirection();

    /**
     * Get the accelerometer for the axis with the given index,
     * where 0 is the X-axis and 1 is the Y-axis.
     *
     * @param axis the axis direction; must be either 0 or 1
     * @return the accelerometer; never null
     * @throws IllegalArgumentException if {@code axis} is invalid
     */
    default public Accelerometer getDirection(int axis) {...}

    /**
     * Get the instantaneous multidimensional acceleration values for all 2 axes.
     *
     * @return the acceleration values for 2 axes; never null
     */
    default public TwoAxisAcceleration getAcceleration() {...}

    /**
     * Create a 2-axis accelerometer from the two individual accelerometers.
     *
     * @param xAxis the accelerometer for the X-axis; may not be null
     * @param yAxis the accelerometer for the Y-axis; may not be null
     * @return the 2-axis accelerometer; never null
     */
    public static TwoAxisAccelerometer create(Accelerometer xAxis,
                                              Accelerometer yAxis) { ...}

}
```

As you can see, this interface is composed of an `Accelerometer` in the x- and y-directions. Each direction's `Accelerometer` is accessed via a dedicated method or via a `getDirection(int)` method that takes an integer as a parameter to specify the desired direction.

The acceleration in both directions can be obtained atomically using the `getAcceleration()` method, which returns an immutable `TwoAxisAcceleration` object with scalar acceleration values (in g's) for each direction, defined as:

```
public class TwoAxisAcceleration {

    /**
     * Get the acceleration in the X-direction.
     * @return the g's along the x-axis
     */
    public double getX() { ... }

    /**
     * Get the acceleration in the Y-direction.
     * @return the g's along the y-axis
     */
    public double getY() { ... }
}
```

Finally, the `TwoAxisAccelerometer` interface defines a static method called `create(…)` that can be used to create a `TwoAxisAccelerometer` object from two individual `Accelerometer` objects:

```
Accelerometer axisX = ...
Accelerometer axisY = ...
TwoAxisAccelerometer accel = TwoAxisAccelerometer.create(axisX,axisY);
```

Or, because `Accelerometer` is a functional interface, we can actually use this method to create a `TwoAxisAccelerometer` object from two method references to methods that take no parameters and return a `double`. For example, given this imaginary class:

```
public class MyHardwareAccelerometer {
    public double getX() {...}
    public double getY() {...}
}
```

then we can create a `TwoAxisAccelerometer` using method references:

```
MyHardwareAccelerometer hw = ......
TwoAxisAccelerometer accel = TwoAxisAccelerometer.create(hw::getX,hw::getY);
```

ThreeAxisAccelerometer

The `org.strongback.components.ThreeAxisAccelerometer` interface provides accelerations along *three* axes. Note that it extends the `TwoAxisAccelerometer` and simply adds a third direction:

```
public interface ThreeAxisAccelerometer extends TwoAxisAccelerometer {

    /**
     * Get the Y-axis accelerometer.
     *
     * @return the accelerometer for the Y-axis; never null
     */
    public Accelerometer getZDirection();

    /**
     * Get the accelerometer for the axis with the given index, where 0 is the X-axis,
     1 is the Y-axis, and 2 is the Z-axis.
     *
     * @param axis the axis direction; must be either 0 or 1
     * @return the accelerometer; never null
     * @throws IllegalArgumentException if {@code axis} is invalid
     */
    default public Accelerometer getDirection(int axis) {...}

    /**
     * Get the instantaneous multidimensional acceleration values for all 3 axes.
     *
     * @return the acceleration values for 3 axes; never null
     */
    default public ThreeAxisAcceleration getAcceleration() {...}

    /**
     * Create a 3-axis accelerometer from the three individual accelerometers.
     *
     * @param xAxis the accelerometer for the X-axis; may not be null
     * @param yAxis the accelerometer for the Y-axis; may not be null
     * @param zAxis the accelerometer for the Z-axis; may not be null
     * @return the 3-axis accelerometer; never null
     */
    public static ThreeAxisAccelerometer create(Accelerometer xAxis,
                                                Accelerometer yAxis,
                                                Accelerometer zAxis) { ...}

    /**
     * Create a 3-axis accelerometer from a 2-axis accelerometer and a separate accelerom
     * @param xAndY the 2-axis accelerometer for the X- and Y-axes; may not be null
     * @param zAxis the accelerometer for the Z-axis; may not be null
     * @return the 3-axis accelerometer; never null
     */
    public static ThreeAxisAccelerometer create(TwoAxisAccelerometer xAndY,
                                                Accelerometer zAxis) {
}
```

As you can see, this interface is composed of an `Accelerometer` in the x-, y-, and z-directions. Each direction's `Accelerometer` is accessed via a dedicated method or via a `getDirection(int)` method that takes an integer as a parameter to specify the desired direction.

The acceleration in all three directions can be obtained atomically using the `getAcceleration()` method, which returns an immutable `ThreeAxisAcceleration` object with scalar acceleration values (in g's) for each direction, defined as:

```
public class ThreeAxisAcceleration extends TwoAxisAcceleration {

    /**
     * Get the acceleration in the Z-direction.
     * @return the g's along the z-axis
     */
    public double getZ() { ... }
}
```

Finally, the `ThreeAxisAccelerometer` interface defines two static method called `create(…)` that can be used to create a `ThreeAxisAccelerometer` object. The first creates one from three individual `Accelerometer` objects:

```
Accelerometer axisX = ...
Accelerometer axisY = ...
Accelerometer axisZ = ...
ThreeAxisAccelerometer accel = ThreeAxisAccelerometer.create(axisX,axisY,axisZ);
```

Because `Accelerometer` is a functional interface, we can actually use this method to create a `ThreeAxisAccelerometer` object from three method references to methods that take no parameters and return a `double`. For example, given this imaginary class:

```
public class MyHardwareAccelerometer {
    public double getX() {...}
    public double getY() {...}
    public double getZ() {...}
}
```

then we can create a `ThreeAxisAccelerometer` using method references:

```
MyHardwareAccelerometer hw = ......
ThreeAxisAccelerometer accel = ThreeAxisAccelerometer.create(hw::getX,hw::getY,hw::getZ);
```

We can also use a variant of the `create(…)` to create a `ThreeAxisAccelerometer` object from a `TwoAxisAccelerometer` and a separate z-axis `Accelerometer` :

```
TwoAxisAccelerometer xy = ...
Accelerometer z = ...
ThreeAxisAccelerometer accel = ThreeAxisAccelerometer.create(xy,z);
```

or using method references or lambdas.

Obtaining hardware accelerometers

Strongback provides a `org.strongback.hardware.Hardware` class with static factory methods for common physical hardware devices:

```
public class Hardware {

    /**
     * Factory method for accelerometers.
     */
    public static final class Accelerometers {
        /**
         * Create a new ThreeAxisAccelerometer for the ADXL345
         * with the desired range using the specified I2C port.
         *
         * @param port the I2C port used by the accelerometer
         * @param range the desired range of the accelerometer
         * @return the accelerometer; never null
         */
        public static ThreeAxisAccelerometer accelerometer(I2C.Port port,
                                                           Range range) { ...}

        /**
         * Create a new ThreeAxisAccelerometer for the ADXL345
         * with the desired range using the specified SPI port.
         *
         * @param port the SPI port used by the accelerometer
         * @param range the desired range of the accelerometer
         * @return the accelerometer; never null
         */
        public static ThreeAxisAccelerometer accelerometer(SPI.Port port,
                                                           Range range) { ... }

        /**
         * Create a new ThreeAxisAccelerometer using the RoboRIO's
         * built-in accelerometer.
         *
         * @return the accelerometer; never null
         */
        public static ThreeAxisAccelerometer builtIn() { ... }
        /**
         * Create a new single-axis Accelerometer using the analog
         * accelerometer on the specified channel, with the given
         * sensitivity and zero value.
         *
         * @param channel the channel for the analog accelerometer
         * @param sensitivity the desired sensitivity in Volts per g
         *        (depends on actual hardware, such as 18mV/g or
         *        '0.018' for ADXL193)
         * @param zeroValueInVolts the voltage that represents no
         *        acceleration (should be determine experimentally)
         * @return the accelerometer; never null
         */
        public static Accelerometer analogAccelerometer(int channel,
                                                        double sensitivity,
                                                        double zeroValueInVolts) { ... }

        /**
         * Create a new single-axis Accelerometer using two analog
         * accelerometer on the specified channel, each with the given
         * sensitivity and zero value.
         *
         * @param xAxisChannel the channel for the X-axis analog accelerometer
         * @param yAxisChannel the channel for the Y-axis analog accelerometer
         * @param sensitivity the desired sensitivity in Volts per G
         *        (depends on actual hardware, such as 18mV/g or
         *        '0.018' for ADXL193)
         * @param zeroValueInVolts the voltage that represents no
         *        acceleration (should be determine experimentally)
         * @return the accelerometer; never null
         */
        public static TwoAxisAccelerometer analogAccelerometer(int xAxisChannel,
                                                               int yAxisChannel,
                                                               double sensitivity,
                                                               double zeroValueInVolts) {
    }

}
```

Obviously you should only use this class in code that will only run on the robot, so we recommend centralizing this logic inside one of your top-level robot classes:

```
public class SimpleRobot extends IterativeRobot {

    private TwoAxisAccelerometer accel;

    @Override
    public void robotInit() {
        ...
        accel = Hardware.Accelerometers.builtin();

        // pass 'accel' to the objects that need it ...
    }
```

You can then pass these objects around to other components that need the accelerometers.

Tip	Testability tip You might think that it’s easy for components to just get the `accel` field on the `SimpleRobot` . However, that embeds knowledge about how to find the accelerometer inside the component. This makes it very difficult to test the component off-robot, because it will always try to get an accelerometer from the `SimpleRobot` object. Instead, it’s much better to use *injection*, which is just a fancy way of saying that when a component is created it should be handed all the objects it needs. So if we have a component that requires an accelerometer, then that component should require the accelerometer is given to it (usually through the constructor, if possible). So, the robot class can create this component and pass the accelerometer to it, while test cases can pass in a mock accelerometer. (We’ll talk about mocks in the next section.) The bottom line is that these other components should depend only upon an *injected* `Accelerometer` , `TwoAxisAccelerometer` , or `ThreeAxisAccelerometer` object, and should never *know* how to find one. This helps keep this component *testable*.

Using accelerometers in tests

Your tests should never use the Hardware class to create accelerometers in your tests. Instead, Strongback provides a series of *mock* classes that can be used in place of the hardware implementations. These mocks extend the normal interfaces as expected, but they add setter methods that your test code can explicitly set the values on these artificial accelerometer objects.

For example, the `MockAccelerometer` class is defined as follows:

```
public class MockAccelerometer implements Accelerometer {

    private volatile double accel;

    @Override
    public double getAcceleration() {
        return accel;
    }

    /**
     * Set the acceleration value {@link #getAcceleration() returned} by this object.
     * @param accel the acceleration value
     * @return this instance to enable chaining methods; never null
     */
    public MockAccelerometer setAcceleration(double accel) {
        this.accel = accel;
        return this;
    }
}
```

It implements `Accelerometer`, but rather than using real hardware it simply has a `accel` field in which it holds the current acceleration. Your tests can change the value of the acceleration at any time using the `setAcceleration(double)` method.

You can create an instance using the no-arg constructor, but we prefer you use the `org.strongback.mock.Mock` class' static factory methods. For example, here's a fragment from a test that creates a mock accelerometer:

```
MockAccelerometer accel = Mock.accelerometer();
accel.setAcceleration(1.1);
// pass accel to a component that needs and uses an Accelerometer
// and verify the component reads the acceleration correctly
```

Strongback provides `MockTwoAxisAccelerometer` and `MockThreeAxisAccelerometer` classes, too, and likewise these can be created via the `org.strongback.mock.Mock` class' static factory methods. Here's a fragment from a test that creates and uses a mock 2-axis accelerometer:

```
MockTwoAxisAccelerometer accel2D = Mock.accelerometer2Axis();
accel2D.getXDirection().setAcceleration(1.1);
accel2D.getYDirection().setAcceleration(1.3);
// pass accel2D to a component that needs and uses a TwoAxisAccelerometer
// and verify the component reads the accelerations correctly
```

And finally here's a fragment from a test that creates and uses a mock 3-axis accelerometer:

```
MockThreeAxisAccelerometer accel3D = Mock.accelerometer3Axis();
accel3D.getXDirection().setAcceleration(1.1);
accel3D.getYDirection().setAcceleration(1.3);
accel3D.getZDirection().setAcceleration(0.95);
// pass accel3D to a component that needs and uses a ThreeAxisAccelerometer
// and verify the component reads the accelerations correctly
```

Angle Sensors

A hardware angle sensor is any device that (surprise!) measures the angle of something relative to something else. Example angle sensors include encoders and potentiometers. Gyroscopes, which directly measure angular velocity (or angular acceleration) can also be used as an angle sensor, although doing so is often very inaccurate because the gyroscope must typically *calculate* angular displacement by integrating angular velocity.

This section describes Strongback's multiple interfaces that represent physical angle sensor devices.

AngleSensor

The `org.strongback.components.AngleSensor` interface represents a simple angle sensor that returns the angular displacement *relative to some _zero point*.

```java
public interface AngleSensor extends Zeroable {

    /**
     * Gets the angular displacement in continuous degrees.
     *
     * @return the positive or negative angular displacement
     */
    public double getAngle();

    /**
     * Change the output so that the current value is considered to be 0
     * @return this object to allow chaining of methods; never null
     */
    @Override
    public AngleSensor zero();

    /**
     * Create an angle sensor around the given function that returns the angle.
     *
     * @param angleSupplier the function that returns the angle; may not be null
     * @return the angle sensor
     */
    public static AngleSensor create(DoubleSupplier angleSupplier) {...}

    /**
     * Creates a new angle sensor that inverts the specified AngleSensor
     * object, so that negative angles become positive angles.
     *
     * @param sensor the {@link AngleSensor} to invert
     * @return an AngleSensor that reads the opposite of the original sensor
     */
    public static AngleSensor invert(AngleSensor sensor) {...}

}
```

The `getAngle()` method returns the angular displacement in *continuous degrees*, meaning the value can be any positive or negative number. Negative angles are assumed to be counter-clockwise and positive values clockwise.

The `zero()` method is inherited from the `Zeroable` interface, and it can be used to set the angle to which subsequent angles will be relative.

The `create(DoubleSupplier)` static factory method will create an `AngleSensor` from any `java.util.function.DoubleSupplier`. The `DoubleSupplier` is a functional interface added in Java 8, and it has a single method that takes no parameters and returns a `double` value. Therefore, `create(DoubleSupplier)` can be used to create an `AngleSensor` from any object, lambda, or method reference that satisfies the same signature (no parameters and returns a `double`). For example, the following fragment creates an AngleSensor that always returns the value `180` :

```
AngleSensor angle = AngleSensor.create(()->180);
```

Or, given some imaginary class `Foo` that has a method that takes no parameters and returns a double:

```
public class Foo {
    ...
    public double bar() {...}
    ...
}
```

the following fragment shows how to create an `AngleSensor` that always uses the `Foo.bar()` method to compute the angle, using a method reference to the `bar()` method on the object `foo` :

```
Foo foo = ...
AngleSensor angle = AngleSensor.create(foo::bar);
```

Finally, the `AngleSensor` also defines the `invert(AngleSensor)` method, which can be used to return a new `AngleSensor` that always inverts the angle returned by another:

```
AngleSensor actual = ...
AngleSensor inverted = AngleSensor.invert(actual);
```

This is far easier than always having to multiply the resulting angle by `-1` !

Compass

The `org.strongback.components.Compass` is an angle sensor that determines the angular displacement in continous degrees. A `Compass` is an `AngleSensor` with an additional method that returns the *relative heading*, which is an angle that is always in the range `[0,360)` . Like `AngleSensor` , it can be zeroed to set the angle at which subsequent angles and headings are based.

```
public interface Compass extends AngleSensor {

    /**
     * Gets the angular displacement of in degrees in the range [0, 360).
     *
     * @return the heading of this {@link Compass}
     */
    public default double getHeading() {...}

    /**
     * Create a angle sensor for the given function that returns the angle.
     *
     * @param angleSupplier the function that returns the angle; may not be null
     * @return the angle sensor
     */
    public static Compass create(DoubleSupplier angleSupplier) {...}
}
```

The `getHeading()` method is a `default` method that is implemented in terms of `AngleSensor.getAngle()` to always return a value greater or equal to `0` but less than `360`.

`Compass` also defines a `create(DoubleSupplier)` method that works the same way as `AngleSensor.create(DoubleSupplier)` except that it returns a `Compass` object instead. This is useful to create a `Compass` backed by custom functionality.

Gyroscope

The `org.strongback.components.Gyroscope` interface represents a device that measures angular velocity (in degrees per second) about a single axis. A gyroscope can indirectly determine angular displacement by integrating velocity with respect to time, which is why it extends `Compass`. Negative values are assumed to be counter-clockwise and positive values are clockwise. Like `AngleSensor`, it can be zeroed to set the angle at which subsequent angles and headings are based.

```
public interface Gyroscope extends Compass {

    /**
     * Gets the rate of change in angle in degrees per second.
     *
     * @return the angular velocity
     */
    public double getRate();

    /**
     * Create a gyroscope for the given functions that returns the anglular
     * displacement and velocity.
     *
     * @param angleSupplier the function that returns the angle; may not be null
     * @param rateSupplier the function that returns the angular acceleration; may not be
     * @return the angle sensor
     */
    public static Gyroscope create(DoubleSupplier angleSupplier,
                                   DoubleSupplier rateSupplier) { ... }

}
```

In addition to the `getHeading()` and `getAngle()` methods inherited from `Compass`, the `getRate()` method is returns the *rate of change* in the angle, otherwise known as the angular velocity. The rate can be positive or negative.

`Gyroscope` also defines a `create(DoubleSupplier,DoubleSupplier)` static factory method for creating a `Gyroscope` given one function that returns the angle and another function that returns the rate. This is useful to create a `Gyroscope` backed by custom functionality.

Obtaining hardware angle sensors

Strongback provides a `org.strongback.hardware.Hardware` class with static factory methods for common physical hardware angle sensing devices:

```
public class Hardware {

    /**
     * Factory method for angle sensors.
     */
    public static final class AngleSensors {

        /**
         * Create a Gyroscope that uses a WPILib Gyro on the specified channel.
         *
         * @param channel the channel the gyroscope is plugged into
         * @return the gyroscope; never null
         */
        public static Gyroscope gyroscope(int channel) {...}

        /**
         * Creates a new AngleSensor from an encoder using the specified
         * channels with the specified distance per pulse.
         *
         * @param aChannel the a channel of the encoder
         * @param bChannel the b channel of the encoder
         * @param distancePerPulse the distance the end shaft spins per pulse
         * @return the angle sensor; never null
         */
        public static AngleSensor encoder(int aChannel,
                                          int bChannel,
                                          double distancePerPulse) {...}

        /**
         * Create a new AngleSensor from a potentiometer on the specified
         * channel and with the given scaling. Since no offset is provided,
         * the resulting angle sensor may often be used with a limit switch
         * to know precisely where a mechanism might be located in space.
         * When the limit switch is triggered, the location is , and
         * the angle sensor can be zeroed at that known position. (See
         * potentiometer(int, double, double) when another switch is
         * not used to help determine the location, and instead the
         * zero point is pre-determined by the physical design of the
         * mechanism.)
         *
         * The scale factor multiplies the 0-1 ratiometric value to return
         * the angle in degrees.
         *
         * For example, let's say you have an ideal 10-turn linear potentiometer
         * attached to a motor attached by chains and a 25x gear reduction
         * to an arm. If the potentiometer (attached to the motor shaft)
         * turned its full 3600 degrees, the arm would rotate 144 degrees.
         * Therefore, the 'fullVoltageRangeToInches' scale factor is
         * '144 degrees / 5 V', or 28.8 degrees/volt.
         *
         * @param channel The analog channel this potentiometer is plugged into.
         * @param fullVoltageRangeToDegrees The scaling factor multiplied by the
         *        analog voltage value to obtain the angle in degrees.
         * @return the angle sensor that uses the potentiometer on the given channel;
         *        never null
         */
        public static AngleSensor potentiometer(int channel,
                                                double fullVoltageRangeToDegrees) {...}

        /**
         * Create a new AngleSensor from an analog potentiometer using the specified
         * channel, scaling, and offset. This method is often used when the offset
         * can be hard-coded by measuring the value of the potentiometer at
         * the mechanism's zero-point. On the other hand, if a limit switch is used
         * to always determine the position of the mechanism upon startup, then see
         * potentiometer(int, double).
         *
         * The scale factor multiplies the 0-1 ratiometric value to return the angle
         * in degrees.
         *
         * For example, let's say you have an ideal 10-turn linear potentiometer
         * attached to a motor attached by chains and a 25x gear reduction to an arm.
         * If the potentiometer (attached to the motor shaft) turned its full 3600
         * degrees, the arm would rotate 144 degrees. Therefore, the
         * 'fullVoltageRangeToInches' scale factor is 144 degrees / 5 V, or
         * 28.8 degrees/volt.
         *
         * To prevent the potentiometer from breaking due to minor shifting in
         * alignment of the mechanism, the potentiometer may be installed with the
         * "zero-point" of the mechanism (e.g., arm in this case) a little ways into
         * the potentiometer's range (for example 30 degrees). In this case, the
         * 'offset' value of '30' is determined from the mechanical design.
         *
         * @param channel The analog channel this potentiometer is plugged into.
         * @param fullVoltageRangeToDegrees The scaling factor multiplied by the
         *        analog voltage value to obtain the angle in degrees.
         * @param offsetInDegrees The offset in degrees that the angle sensor will
         *        subtract from the underlying value before returning the angle
         * @return the angle sensor that uses the potentiometer on the given channel;
         *        never null
         */
        public static AngleSensor potentiometer(int channel,
                                                double fullVoltageRangeToDegrees,
                                                double offsetInDegrees) {...}
    }

}
```

Obviously you should only use this class in code that will only run on the robot, so we recommend centralizing this logic inside one of your top-level robot classes:

```
public class SimpleRobot extends IterativeRobot {

    private Gyroscope gyro;

    @Override
    public void robotInit() {
        ...
        gyro = Hardware.AngleSensors.gyroscope(3);

        // pass 'gyro' to the objects that need it ...
    }
```

You can then pass these angle sensor, compasse, or gyroscope objects around to other components that need them.

Tip	Testability tip You might think that it's easy for components to just get the `gyro` field on the `SimpleRobot`, perhaps via a getter method. However, that embeds knowledge about how to find the sensor inside the component that uses it. This makes it very difficult to test the component off-robot, because it will always try to get a hardware-based sensor from the `SimpleRobot` object. Instead, it's much better to use *injection*, which is just a fancy way of saying that when a component is created it should be handed all the objects it needs. So if we have a component that requires a sensor, then that component should require the sensor be given to it (usually through the constructor, if possible). The robot class can create this component and pass the sensor to it, while test cases can pass in a mock sensor. (We'll talk about mocks in the next section.) The bottom line is that these other components should depend only upon an *injected* `AngleSensor`, `Compass`, or `Gyroscope` objects, and should never *know* how to find one. This helps keep these other components *testable*.

Using angle sensors in tests

Your tests should never use the Hardware class to create angle sensors, compasses, or gyroscopes in your tests. Instead, Strongback provides a series of *mock* classes that can be used in place of the hardware implementations. These mocks extend the normal interfaces as expected, but they add setter methods that your test code can explicitly set the angles and/or rates on these artificial mock objects.

For example, the `MockAngleSensor` class is defined as follows:

```
public class MockAngleSensor extends MockZeroable implements AngleSensor {

    @Override
    public MockAngleSensor zero() {
        super.zero();
        return this;
    }

    @Override
    public double getAngle() {
        return super.getValue();
    }

    /**
     * Set the angle value {@link #getAngle() returned} by this object.
     *
     * @param angle the angle value
     * @return this instance to enable chaining methods; never null
     */
    public MockAngleSensor setAngle(double angle) {
        super.setValue(angle);
        return this;
    }
}
```

where `MockZeroable` manages the scalar value and zeroing functionality, and is defined as:

```
abstract class MockZeroable implements Zeroable {

    private volatile double zero = 0;
    private volatile double value;

    protected double getValue() {
        return value - zero;
    }

    protected void setValue( double value) {
        this.value = value;
    }

    @Override
    public MockZeroable zero() {
        zero = value;
        return this;
    }
}
```

The `MockAngleSensor` implements `AngleSensor`, but rather than using real hardware it simply uses a field to track the current angle. Your tests can change the value of the acceleration at any time using the `setAngle(double)` method, and it can zero the sensor at any time using the `zero()` method.

Strongback provides `MockCompass` and `MockGyroscope` classes, too.

You can create mock objects using the `org.strongback.mock.Mock` class' static factory methods. For example, here's a fragment from a test that creates a mock compass:

```
MockCompass compass = Mock.compass();
compass.setAngle(721.1);
// pass compass to a component that needs and uses a Compass or AngleSensor
// and verify the component correctly reads the angle and/or heading
```

Distance Sensors

A hardware distance sensor is any device that measures the distance of something relative to something else. Example distance sensors include digital or analog ultrasonic sensors and potentiometers.

This section describes Strongback's interface that represent physical distance sensor devices.

DistanceSensor

The `org.strongback.components.DistanceSensor` interface represents a simple distance sensor that returns the distance *relative to some _zero point*.

```
public interface DistanceSensor extends Zeroable {

    /**
     * Gets the current distance, in inches, relative to the zero-point.
     *
     * @return the distance in inches
     */
    public double getDistanceInInches();

    /**
     * Gets the current distance, in feet, relative to the zero-point.
     *
     * @return the distance in feet
     */
    default public double getDistanceInFeet() {...}

    /**
     * Use the current distance value as the new zero-point,
     * to which all subsequent measurements will be relative.
     *
     * @return this object to allow chaining of methods; never null
     */
    @Override
    default public DistanceSensor zero() {...}

    /**
     * Create a distance sensor for the given function that returns the distance.
     *
     * @param distanceSupplier the function that returns the distance; may not be null
     * @return the angle sensor
     */
    public static DistanceSensor create(DoubleSupplier distanceSupplier) {...}

    /**
     * Inverts the specified DistanceSensor so that negative distances
     * become positive distances.
     *
     * @param sensor the DistanceSensor to invert
     * @return the new DistanceSensor that reads the negated distance
     * of the original sensor
     */
    public static DistanceSensor invert(DistanceSensor sensor) {

}
```

The `getDistanceInInches()` method returns the relative distance in *inches*, meaning the value can be any positive or negative number based upon the zero-point. The `getDistanceInFeet()` default method is implemented in terms of the `getDistanceInInches()` method, making `DistanceSensor` a functional interface.

The `zero()` method is inherited from the `Zeroable` interface, and it can be used to set the distance to which subsequent distances will be relative.

The `create(DoubleSupplier)` static factory method will create a `DistanceSensor` from any `java.util.function.DoubleSupplier`. The `DoubleSupplier` is a functional interface added in Java 8, and it has a single method that takes no parameters and returns a `double` value. Therefore, `create(DoubleSupplier)` can be used to create a `DistanceSensor` from any object, lambda, or method reference that satisfies the same signature (no parameters and returns a `double`). For example, the following fragment creates an DistanceSensor that always returns the value `100`:

```
DistanceSensor dist = DistanceSensor.create(()->100);
```

Or, given some imaginary class `Foo` that has a method that takes no parameters and returns a double:

```
public class Foo {
    ...
    public double bar() {...}
    ...
}
```

the following fragment shows how to create a `DistanceSensor` that always uses the `Foo.bar()` method to compute the distance, using a method reference to the `bar()` method on the object `foo`:

```
Foo foo = ...
DistanceSensor dist = DistanceSensor.create(foo::bar);
```

Finally, the `AngleSensor` also defines the `invert(AngleSensor)` method, which can be used to return a new `AngleSensor` that always inverts the angle returned by another:

```
DistanceSensor actual = ...
DistanceSensor inverted = DistanceSensor.invert(actual);
```

This is far easier than always having to multiply the resulting distance by `-1`!

Obtaining hardware angle sensors

Strongback provides a `org.strongback.hardware.Hardware` class with static factory methods for common physical hardware angle sensing devices:

```
public class Hardware {
    ...

    /**
     * Factory method for distance sensors.
     */
    public static final class DistanceSensors {
        /**
         * Create a digital ultrasonic distance sensor for an ultrasonic
         * sensor that uses the specified channels.
         *
         * @param pingChannel the digital output channel to use for sending pings
         * @param echoChannel the digital input channel to use for receiving echo respons
         * @return a DistanceSensor linked to the specified channels
         */
        public static DistanceSensor digitalUltrasonic(int pingChannel,
                                                       int echoChannel) {...}

        /**
         * Create an analog distance sensor for an analog input sensor using
         * the specified channel.
         *
         * @param channel the channel the sensor is connected to
         * @param voltsToInches the conversion from analog volts to inches
         * @return a {@link DistanceSensor} linked to the specified channel
         */
        public static DistanceSensor analogUltrasonic(int channel,
                                                      double voltsToInches) {...}

        /**
         * Create a new distance sensor from an analog potentiometer using
         * the specified channel and scaling. Since no offset is provided,
         * the resulting distance sensor may often be used with a limit switch
         * to know precisely where a mechanism might be located in physical space,
         * and the distance sensor can be zeroed at that position.
         * (See potentiometer(int, double, double) when a switch is not used to
         * help determine the location, and instead the zero point is
         * pre-determined by the physical design of the mechanism.)
         *
         * The scale factor multiplies the 0-1 ratiometric value to return
         * useful units. Generally, the most useful scale factor will be the
         * angular or linear full scale of the potentiometer.
         *
         * For example, let's say you have an ideal single-turn linear
         * potentiometer attached to a robot arm. This pot will turn 270 degrees
         * over the 0V-5V range while the end of the arm travels 20 inches.
         * Therefore, the `fullVoltageRangeToInches` scale factor is
         * 20 inches / 5 V, or 4 inches/volt.
         *
         * @param channel The analog channel this potentiometer is plugged into.
         * @param fullVoltageRangeToInches The scaling factor multiplied by
         *        the analog voltage value to obtain inches.
         * @return the distance sensor that uses the potentiometer on the
         *        given channel; never null
         */
        public static DistanceSensor potentiometer(int channel,
                                                   double fullVoltageRangeToInches) {...}

        /**
         * Create a new distance sensor from an analog potentiometer using
         * the specified channel, scaling and offset. This method is often
         * used when the offset can be hard-coded by first measuring the
         * value of the potentiometer. On the other hand, if a limit switch
         * is used to always determine the position of the mechanism upon
         * startup, then see potentiometer(int, double).
         *
         * The scale factor multiplies the 0-1 ratiometric value to return
         * useful units, and the offset is added after the scaling. Generally,
         * the most useful scale factor will be the angular or linear full
         * scale of the potentiometer.
         *
         * For example, let's say you have an ideal single-turn linear
         * potentiometer attached to a robot arm. This pot will turn 270 degrees
         * over the 0V-5V range while the end of the arm travels 20 inches.
         * Therefore, the `fullVoltageRangeToInches` scale factor is
         * 20 inches / 5 V, or 4 inches/volt.
         *
         * To prevent the potentiometer from breaking due to minor shifting
         * in alignment of the mechanism, the potentiometer may be installed
         * with the "zero-point" of the mechanism (e.g., arm in this case)
         * a little ways into the potentiometer's range (for example 10 degrees).
         * In this case, the `offset` value is measured from the physical
         * mechanical design and can be specified to automatically remove
         * the 10 degrees from the potentiometer output.
         *
         * @param channel The analog channel this potentiometer is plugged into.
         * @param fullVoltageRangeToInches The scaling factor multiplied by
         *        the analog voltage value to obtain inches.
         * @param offsetInInches The offset in inches that the distance sensor
         *        will subtract from the underlying value before
         *        returning the distance
         * @return the distance sensor that uses the potentiometer on the
         *        given channel; never null
         */
        public static DistanceSensor potentiometer(int channel,
                                                   double fullVoltageRangeToInches,
                                                   double offsetInInches) {...}
    }
    ...
}
```

Obviously you should only use this class in code that will only run on the robot, so we recommend centralizing this logic inside one of your top-level robot classes:

```
public class SimpleRobot extends IterativeRobot {

    private DistanceSensor dist;

    @Override
    public void robotInit() {
        ...
        dist = Hardware.DistanceSensors.analogUltrasonic(3,4.0);

        // pass 'dist' to the objects that need it ...
    }
```

You can then pass the distance sensor reference around to other components that need them.

Tip	Testability tip You might think that it's easy for components to just get the `dist` field on the `SimpleRobot`, perhaps via a getter method. However, that embeds knowledge about how to find the sensor inside the component that uses it. This makes it very difficult to test the component off-robot, because it will always try to get a hardware-based sensor from the `SimpleRobot` object. Instead, it's much better to use *injection*, which is just a fancy way of saying that when a component is created it should be handed all the objects it needs. So if we have a component that requires a sensor, then that component should require the sensor be given to it (usually through the constructor, if possible). The robot class can create this component and pass the sensor to it, while test cases can pass in a mock sensor. (We'll talk about mocks in the next section.) The bottom line is that these other components should depend only upon *injected* `DistanceSensor` objects, and should never *know* how to find one. This helps keep these other components *testable*.

Using distance sensors in tests

Your tests should never use the Hardware class to create distance sensors in your tests. Instead, Strongback provides a series of *mock* classes that can be used in place of the hardware implementations. These mocks extend the normal `DistanceSensor` interface as expected, but they add setter methods that your test code can explicitly set the distance value returned by the sensor.

The `MockDistanceSensor` class is defined as follows:

```
public class MockDistanceSensor extends MockZeroable implements DistanceSensor {

    @Override
    public MockDistanceSensor zero() {
        super.zero();
        return this;
    }

    @Override
    public double getDistanceInInches() {
        return super.getValue();
    }

    /**
     * Set the distance in inches return by this object.
     *
     * @param distance the new distance in inches
     * @return this instance to enable chaining methods; never null
     * @see #setDistanceInFeet(double)
     */
    public MockDistanceSensor setDistanceInInches(double distance) {...}

    /**
     * Set the distance in feet return by this object.
     *
     * @param distance the new distance in feet
     * @return this instance to enable chaining methods; never null
     * @see #setDistanceInInches(double)
     */
    public MockDistanceSensor setDistanceInFeet(double distance) {
        return setDistanceInInches(distance * 12.0);
    }
```

where `MockZeroable` manages the scalar value and zeroing functionality, and is defined as:

```
abstract class MockZeroable implements Zeroable {

    private volatile double zero = 0;
    private volatile double value;

    protected double getValue() {
        return value - zero;
    }

    protected void setValue( double value) {
        this.value = value;
    }

    @Override
    public MockZeroable zero() {
        zero = value;
        return this;
    }
}
```

You can create mock objects using the `org.strongback.mock.Mock` class' static factory methods. For example, here's a fragment from a test that creates a mock compass:

```
MockDistanceSensor dist = Mock.distanceSensor();
dist.setDistanceInInches(21.1);
// pass compass to a component that needs and uses a DistanceSensor
// and verify the component correctly reads the distance
```

Switches

A hardware switch is any device that have an active state when triggered and an inactive state when not triggered. Conceptually switches are relatively abstract, but can represent multiple kinds of hardware devices, including Reed switches, limit switches, proximity sensors, buttons, and magnetic switches. Fuses are a specialization of a switch that can be programmatically reset. Switches can also be used to represent abstract components that are on or off.

This section describes Strongback's multiple interfaces that represent physical switches and fuses.

Switch

The `org.strongback.components.Switch` interface represents a simple device that exists in one of two states: triggered or not triggered. It has one abstract method, and therefore is a functional interface in Java 8.

```
public interface Switch {
    /**
     * Checks if this switch is triggered.
     *
     * @return true if this switch was triggered, or false otherwise
     */
    public boolean isTriggered();

    /**
     * Create a switch that is always triggered.
     * @return the always-triggered switch; never null
     */
    public static Switch alwaysTriggered() {...}

    /**
     * Create a switch that is never triggered.
     * @return the never-triggered switch; never null
     */
    public static Switch neverTriggered() {...}
}
```

The `isTriggered()` method returns `true` if triggered or `false` otherwise.

Switches are so simple that they don't even need a static factory method, since any lambda or method reference to a method that returns a boolean can be used as `Switch`.

```
Switch swtch = ()->true;
```

Or, given some imaginary class `Foo` that has a method that takes no parameters and returns a boolean:

```
public class Foo {
    ...
    public boolean bar() {...}
    ...
}
```

and a method that takes a switch:

```
public interface DataRecorder {
    ...
    public DataRecorder register(String name, Switch swtch);
    ...
}
```

the following fragment shows how to use the `Foo` instance as a switch:

```
DataRecorder recorder = ...
Foo foo = ...
recorder.register("button",foo::bar);
```

Fuse

The `org.strongback.components.Fuse` interface is a specialization of a `Switch` that allows the triggered state to be (re)set:

```
public interface Fuse extends Switch {
    /**
     * Trigger the fuse. Once this method is called, the `isTriggered()`
     * method switches but then will never change until it is `reset()`.
     *
     * @return this object to allow for chaining methods together; never null
     */
    public Fuse trigger();

    /**
     * Reset the fuse so it is no longer triggered.
     *
     * @return this object to allow for chaining methods together; never null
     */
    public Fuse reset();

    /**
     * Create a simple fuse that is manually triggered and manually reset().
     *
     * @return the fuse; never null
     */
    public static Fuse create() {...}

    /**
     * Create a fuse that can be manually reset but that will automatically
     * reset after the given delay.
     *
     * @param delay the time after the fuse is triggered that it should
     *        automatically reset; must be positive
     * @param unit the time units for the delay; may not be null
     * @param clock the clock that the fuse should use; if null, the system
     *        clock will be used
     * @return the auto-resetting fuse; never null
     */
    public static Fuse autoResetting(long delay, TimeUnit unit, Clock clock) {...}
}
```

The `trigger()` method changes the fuse's state to be *triggered*. Only after `reset()` is called will the state change to *not-triggered*.

Other than physical fuses, you can use the static factory method `create()` to create fuses that operate entirely manually:

```
Fuse fuse = Fuse.create();
assert !fuse.isTriggered();

// manually trigger the fuse
fuse.trigger();
assert fuse.isTriggered();

// manually reset the fuse
fuse.reset();
assert !fuse.isTriggered();
```

How might you use a manual fuse? Really, you can use it anywhere you need to maintain or represent a boolean state. For example, your robot might have two modes that can be changed based upon commands or some sensor state, and you can represent this easily with a manual fuse that both the command and sensor can manually alter.

Strongback also has an *automatically resetting* fuse. You can also use the static factory method `autoResetting(…)` to create a fuse that can be manually triggered and reset, but that will automatically reset after a specified delay following it being manually triggered. Here's a non-realistic example:

```
Clock clock = Clock.system();
Fuse fuse = Fuse.autoResetting(5,TimeUnit.SECONDS,clock);
assert !fuse.isTriggered();

// manually trigger the fuse
fuse.trigger();
assert fuse.isTriggered();

Thread.sleep(5010);
assert !fuse.isTriggered();
```

How might you use an automatic fuse? One example might be to use an automatic fuse to model a physical mechanism that after being used requires 4 seconds to "recharge" before it can be used again:

```
Fuse mechanism = Fuse.autoResetting(4,TimeUnit.SECONDS,clock);
...
if ( mechanism.isTriggered() ) {
   // Still not ready ...
   ...
}
```

Obtaining hardware switches and fuses

Strongback provides a `org.strongback.hardware.Hardware` class with static factory methods for common physical hardware switches:

```
public class Hardware {
    ...

    /**
     * Factory method for different kinds of switches.
     */
    public static final class Switches {

        /**
         * Create a relay on the specified channel.
         *
         * @param channel the channel the relay is connected to
         * @return a relay on the specified channel
         */
        public static Relay relay(int channel) {...}

        /**
         * Create a generic normally closed digital switch sensor on the
         * specified digital channel.
         *
         * @param channel the channel the switch is connected to
         * @return a switch on the specified channel
         */
        public static Switch normallyClosed(int channel) {...}

        /**
         * Create a generic normally open digital switch sensor on the
         * specified digital channel.
         *
         * @param channel the channel the switch is connected to
         * @return a switch on the specified channel
         */
        public static Switch normallyOpen(int channel) {...}

        /**
         * Option for analog switches.
         *
         * @see #analog(int, double, double, AnalogOption, TriggerMode)
         */
        public static enum AnalogOption {
            /**
             * The filtering option of the analog trigger uses a 3-point
             * average reject filter. This filter uses a circular
             * buffer of the last three data points and selects the
             * outlier point nearest the median as the output. The primary
             * use of this filter is to reject data points which errantly
             * (due to averaging or sampling) appear within the
             * window when detecting transitions using the Rising Edge
             * and Falling Edge functionality of the analog trigger
             */
            FILTERED,
            /**
             * The analog output is averaged and over sampled.
             */
            AVERAGED,
            /**
              * No filtering or averaging is to be used.
              */
            NONE;
        }

        /**
         * Trigger mode for analog switches.
         *
         * @see #analog(int, double, double, AnalogOption, TriggerMode)
         */
        public static enum TriggerMode {
            /**
             * The switch is triggered only when the analog value
             * is inside the range, and not triggered if it is
             * outside (above or below)
             */
            IN_WINDOW,
            /**
             * The switch is triggered only when the value is above
             * the upper limit, and not triggered if it is below
             * the lower limit and maintains the previous state
             * if in between (hysteresis)
             */
            AVERAGED;
        }

        /**
         * Create an analog switch sensor that is triggered when the
         * value exceeds the specified upper voltage and that is no
         * longer triggered when the value drops below the specified
         * lower voltage.
         *
         * @param channel the port to use for the analog trigger 0-3
         *        are on-board, 4-7 are on the MXP port
         * @param lowerVoltage the lower voltage limit that below
         *        which will result in the switch no longer being triggered
         * @param upperVoltage the upper voltage limit that above which
         *        will result in triggering the switch
         * @param option the trigger option; may not be null
         * @param mode the trigger mode; may not be null
         * @return the analog switch; never null
         */
        public static Switch analog(int channel,
                                    double lowerVoltage,
                                    double upperVoltage,
                                    AnalogOption option,
                                             TriggerMode mode) {...}
    }
    ...
}
```

Obviously you should only use this class in code that will only run on the robot, so we recommend centralizing this logic inside one of your top-level robot classes:

```
public class SimpleRobot extends IterativeRobot {

    private Switch retracted;
    private Switch extended;

    @Override
    public void robotInit() {
        ...
        // limit switches in channels 3 and 4
        retracted = Hardware.Switches.normallyOpen(3);
        extended = Hardware.Switches.normallyOpen(4);

        // pass 'retracted' and/or 'extended` to the objects that need it ...
    }
```

You can then pass the switches around to other components that need them.

Tip	Testability tip You might think that it's easy for components to just get the `extended` or `retracted` fields on the `SimpleRobot`, perhaps via getter methods. However, that embeds knowledge about how to find the sensors inside the component that uses it. This makes it very difficult to test the component off-robot, because it will always try to get a hardware-based sensor from the `SimpleRobot` object. Instead, it's much better to use *injection*, which is just a fancy way of saying that when a component is created it should be handed all the objects it needs. So if we have a component that requires a sensor, then that component should require the sensor be given to it (usually through the constructor, if possible). The robot class can create this component and pass the sensor to it, while test cases can pass in a mock sensor. (We'll talk about mocks in the next section.) The bottom line is that these other components should depend only upon *injected* `Switch` or `Fuse` objects, and should never *know* how to find one. This helps keep these other components *testable*.

Using switches in tests

Your tests should never use the Hardware class to create switches or fuses in your tests. Instead, Strongback provides a series of *mock* classes that can be used in place of the hardware implementations. These mocks extend the normal `Switch` interface as expected, but they add setter methods that your test code can explicitly set whether a mock switch is triggered.

The `MockSwitch` class is defined as follows:

```
public class MockSwitch implements Switch {

    private boolean triggered = false;

    @Override
    public boolean isTriggered() {
        return triggered;
    }

    /**
     * Set whether this switch is to be triggered.
     * @param triggered true if the switch is to be triggered, or false otherwise
     * @return this object to allow chaining of methods; never null
     */
    public MockSwitch setTriggered( boolean triggered ) {
        this.triggered = triggered;
        return this;
    }

    /**
     * Set this switch as being triggered.
     * @return this object to allow chaining of methods; never null
     */
    public MockSwitch setTriggered() {
        setTriggered(true);
        return this;
    }

    /**
     * Set this switch as being not triggered.
     * @return this object to allow chaining of methods; never null
     */
    public MockSwitch setNotTriggered() {
        setTriggered(false);
        return this;
    }

    @Override
    public String toString() {
        return triggered ? "closed" : "open";
    }
}
```

You can create mock objects using the `org.strongback.mock.Mock` class' static factory methods. For example, here’s a fragment from a test that creates a few mocks switches with different initial states:

```
MockSwitch s1 = Mock.notTriggeredSwitch();
MockSwitch s2 = Mock.triggeredSwitch();
assert !s1.isTriggered();
assert s2.isTriggered();

s2.setNotTriggered();
assert !s2.isTriggered();
```

Strongback does not provide a `MockFuse` , since `Fuse` already has methods to set the state. You can, however, create "mock" fuses using the `Mock` static factory methods:

```
Fuse f1 = Mock.notTriggeredFuse();
Fuse f2 = Mock.triggeredFuse();
assert !f1.isTriggered();
assert f2.isTriggered();

f2.reset();
assert !f2.isTriggered();
```

Motors

From Strongback’s perspective, a motor is a device that can be set to operate at a controllable speed. This is intentionally a very abstract concept. Each motor object represents a *physical motor and its motor controller*. For example, a CIM motor and Talon SRX controller devices can be modeled in Strongback as a single `Motor` object.

This section goes into more detail about the Strongback interfaces that are related to motors.

SpeedSensor

The `org.strongback.components.SpeedSensor` interface represents a simple sensor that can detect speed:

```
public interface SpeedSensor {

    /**
     * Gets the current speed.
     *
     * @return the speed
     */
    public double getSpeed();
}
```

There is only one abstract method, so `SpeedSensor` is a functional interface. Although FRC doesn’t normally have sensors that directly measure speed, it still is an independent concept worth having. Besides, you might come up with your own speed sensor implementations: for exmaple, your robot might calculate the (approximate) speed by measuring rotations per second on an encoder, and you could do this with a custom implementation of `SpeedSensor`.

SpeedController

The `org.strongback.components.SpeedController` interface can control (or set) the speed of a device:

```
public interface SpeedController {

    /**
     * Sets the speed.
     *
     * @param speed the new speed as a double
     * @return this object to allow chaining of methods; never null
     */
    public SpeedController setSpeed(double speed);
}
```

There is only one abstract method, so `SpeedController` is a functional interface. Again, this interface may not be terribly useful on its own, but it still is an independent concept worth having.

Stoppable

The `org.strongback.components.Stoppable` interface represents any device that can be stopped, and is defined with a very simple functional interface:

```
public interface Stoppable {

    /**
     * Stops this component.
     */
    public void stop();
}
```

Motor

The `org.strongback.components.Motor` interface extends `SpeedSensor` , `SpeedController` , and `Stoppable` (although it overrides many methods to downcast the return type or provide documentation):

```
public interface Motor extends SpeedSensor, SpeedController, Stoppable {

    /**
     * Gets the current speed.
     *
     * @return the speed, will be between -1.0 and 1.0 inclusive
     */
    @Override
    public double getSpeed();

    /**
     * Sets the speed of this motor.
     *
     * @param speed the new speed as a double, clamped to -1.0 to 1.0 inclusive
     * @return this object to allow chaining of methods; never null
     */
    @Override
    public Motor setSpeed(double speed);

    /**
     * Stops this motor. Same as calling `setSpeed(0.0)`.
     */
    @Override
    default public void stop() {
        setSpeed(0.0);
    }

    public enum Direction {
        FORWARD, REVERSE, STOPPED
    }

    /**
     * Gets the current direction of this motor, which can be
     * Direction.FORWARD, Direction.REVERSE, or Direction.STOPPED.
     *
     * @return the direction of this motor; never null
     */
    default public Direction getDirection() {...}

    /**
     * Create a new motor that inverts this motor.
     *
     * @return the new inverted motor; never null
     */
    default Motor invert() {...}

    /**
     * Create a new motor instance that is actually composed
     * of two other motors that will be controlled identically.
     * This is useful when multiple motors are controlled in
     * the same way, such as on one side of a TankDrive.
     *
     * @param motor1 the first motor, and the motor from which
     *        the speed is read; may not be null
     * @param motor2 the second motor; may not be null
     * @return the composite motor; never null
     */
    static Motor compose(Motor motor1,
                         Motor motor2) {...}

    /**
     * Create a new motor instance that is actually composed
     * of three other motors that will be controlled identically.
     * This is useful when multiple motors are controlled in
     * the same way, such as on one side of a TankDrive.
     *
     * @param motor1 the first motor, and the motor from which
     *        the speed is read; may not be null
     * @param motor2 the second motor; may not be null
     * @param motor3 the third motor; may not be null
     * @return the composite motor; never null
     */
    static Motor compose(Motor motor1,
                         Motor motor2,
                         Motor motor3) {...}

    /**
     * Create a new motor instance that inverts the speed sent
     * to and read from another motor. This is useful on
     * TankDrive, where all motors on one side are physically
     * inverted compared to the motors on the other side.
     *
     * @param motor the motor to invert; may not be null
     * @return the inverted motor; never null
     */
    static Motor invert(Motor motor) {...}
}
```

The `Motor` interface is pretty straightforward with its `getSpeed()`, `setSpeed(double)`, and `stop()` methods. It also defines a `getDirection()` method that returns `Direction.FORWARD` when its speed is positive, `Direction.REVERSE` when its speed is negative, and `Direction.STOPPED` when its speed is 0.

Interestingly, the `Motor` interface also provides a number of static factory methods that make it very easy to compose multiple `Motor` instances together so they can be treated as a single `Motor` instance. This is very common on some types of chassis, where multiple motors are ganged together on a single gearbox. For example, this fragment shows how to create the `Motor` objects for a tank-style chassis that has 2 motors on each side:

```
Motor leftFront = ...
Motor leftRear = ...
Motor rightFront = ...
Motor rightRear = ...
Motor left = Motor.compose(leftFront,leftRear);
Motor right = Motor.compose(rightFront,rightRear);
DriveTrain drive = TankDrive.create(left, right.invert());
```

We'll see later on how your robot can obtain the four `Motor` objects that represent the four pairs of hardware motors and motor controller devices. But you can see how easy it is to make `left` control both the `leftFront` and `leftRear` motors. The code is concise and very easy to read and understand.

The example also shows the use of the `invert()` method, which creates a new `Motor` whose speed is the negation of the speed of the `Motor` on which the method is called. (For convenience, there is also a static form of the same method.)

LimitedMotor

The `org.strongback.components.LimitedMotor` interface is a specialization of `Motor`. It represents a motor that moves a component that is limited on the two extremes, and that should automatically stop when the `Switch` sensor on either side is triggered. A `LimitedMotor` also tracks the position of the component in relation to the two limits.

```
public interface LimitedMotor extends Motor {

    /**
     * The possible positions for a limited motor.
     */
    public enum Position {
        /**
         * The motor is at the forward direction limit.
         **/
        FORWARD_LIMIT,
        /**
         * The motor is at the reverse direction limit.
         **/
        REVERSE_LIMIT,
        /**
         * The motor is between the forward and reverse limits,
         * but the exact position is unknown.
         **/
        UNKNOWN
    }

    @Override
    public LimitedMotor setSpeed(double speed);

    /**
     * Get the switch that signals when this motor reaches its limit
     * in the forward direction.
     *
     * @return the forward direction limit switch; never null
     */
    public Switch getForwardLimitSwitch();

    /**
     * Get the switch that signals when this motor reaches its limit
     * in the reverse direction.
     *
     * @return the reverse direction limit switch; never null
     */
    public Switch getReverseLimitSwitch();
```

```
/**
 * Tests if this limited motor is at the high limit. This is equivalent
 * to calling `getForwardLimitSwitch().isTriggered()`.
 *
 * @return true if this motor is at the forward limit; or false otherwise
 */
default public boolean isAtForwardLimit() {
    return getForwardLimitSwitch().isTriggered();
}

/**
 * Tests if this limited motor is at the low limit. This is equivalent
 * to calling `getReverseLimitSwitch().isTriggered()`.
 *
 * @return true if this motor is at the reverse limit; or false otherwise
 */
default public boolean isAtReverseLimit() {
    return getReverseLimitSwitch().isTriggered();
}

/**
 * Moves this limited towards the forward limit. This method should
 * be called once per loop until the movement is completed.
 *
 * @param speed the speed at which the underlying motor should spin
 *        in the forward direction
 * @return true if the motor remains moving, or false if it has
 *        reached the forward limit
 */
default public boolean forward(double speed) {...}

/**
 * Moves this {@link LimitedMotor} towards the reverse limit.
 * This method should be called once per loop until the movement
 * is completed.
 *
 * @param speed the speed at which the underlying motor should spin
 *        in the reverse direction
 * @return true if the motor remains moving, or false if it has
 *        reached the forward limit
 */
default public boolean reverse(double speed) {...}

/**
 * Gets the current position of this limited motor. Can be
 * HIGH, LOW, or UNKNOWN.
 *
 * @return the current position of this motor
 */
default public Position getPosition() {...}

/**
```

```
     * Create a limited motor around the given motor and switches.
     *
     * @param motor the {@link Motor} being limited; may not be null
     * @param forwardSwitch the {@link Switch} that signals the motor
     *        reached its limit in the forward direction, or null if
     *        there is no limit switch
     * @param reverseSwitch the {@link Switch} that signals the motor
     *        reached its limit in the reverse direction, or null if
     *        there is no limit switch
     * @return the limited motor; never null
     * @throws IllegalArgumentException if the `motor` parameter is null
     */
    public static LimitedMotor create(Motor motor,
                                      Switch forwardSwitch,
                                      Switch reverseSwitch) {...}
}
```

The important thing to understand is that a `LimitedMotor` doesn't check the limit switches in a background thread. Instead, it only does this when `forward` or `reverse` methods are called. Therefore, these methods should be called in the run loop of the robot, and when one of these method invocations detects that a limit has been reached it will automatically `stop()` itself.

This class works very well in the commmand framework. Here, you define a command that tells the motor to move forward (or reverse) at certain speed, and that stops as soon as `isAtForwardLimit()` (or `isAtReverseLimit()`) returns true. Here's a very concise way to do that:

```
Motor motor = ...
Switch forwardLimit = ...
Switch reverseLimit = ...
LimitedMotor limitedMotor = LimitedMotor.create(motor,forwardLimit,reverseLimit);

// Begin moving forward at 1/2 speed until limit is reached ...
Strongback.submit(Command.create(()->limitedMotor.forward(0.5));
```

We'll see more examples like this in the section on the commmand framework. But note that the `limitedMotor.forward(0.5)` tells the `LimitedMotor` object to move forward at half-speed, and since it returns a boolean as to whether it is still moving, this lambda is sufficient for the `Command` to know how long to keep running.

TalonSRX

The `org.strongback.components.TalonSRX` interface is a specialization of `LimitedMotor` . The Talon SRX is an advanced motor controller that can be wired with external limit switches to automatically stop the forward and reverse directions when the limit switches are triggered. It also has a built-in current sensor and position (angle) sensor. All of this is handled within the Talon SRX hardware.

```
public interface TalonSRX extends LimitedMotor {

    @Override
    public TalonSRX setSpeed(double speed);

    /**
     * Get the angle sensor (encoder) hooked up to the Talon SRX
     * motor controller.
     *
     * @return the angle sensor; never null, but if not hooked up
     *         the sensor will always return a meaningless value
     */
    public AngleSensor getAngleSensor();

    /**
     * Get the Talon SRX's current sensor.
     *
     * @return the current sensor; never null
     */
    public CurrentSensor getCurrentSensor();

}
```

Obtaining hardware motors

Strongback provides a `org.strongback.hardware.Hardware` class with static factory methods for obtaining various motors based upon the kind of physical motor controller device:

```
public class Hardware {
    ...

    /**
     * Factory method for different kinds of motors.
     */
    public static final class Motors {

        /**
         * Create a motor driven by a Talon speed controller on the
         * specified channel. The speed output is limited to [-1.0,1.0]
         * inclusive.
         *
         * @param channel the channel the controller is connected to
         * @return a motor on the specified channel
         */
        public static Motor talon(int channel) {...}

        /**
         * Create a motor driven by a Talon speed controller on the
         * specified channel, with a custom speed limiting function.
         *
         * @param channel the channel the controller is connected to
         * @param speedLimiter function that will be used to limit
         *        the speed; may not be null
         * @return a motor on the specified channel
         */
        public static Motor talon(int channel,
                                  DoubleToDoubleFunction speedLimiter) {...}

        /**
         * Create a motor driven by a Jaguar speed controller on the
         * specified channel. The speed output is limited to [-1.0,1.0]
         * inclusive.
         *
         * @param channel the channel the controller is connected to
         * @return a motor on the specified channel
         */
        public static Motor jaguar(int channel) {...}

        /**
         * Create a motor driven by a Jaguar speed controller on the
         * specified channel, with a custom speed limiting function.
         *
         * @param channel the channel the controller is connected to
         * @param speedLimiter function that will be used to limit
         *        the speed; may not be null
         * @return a motor on the specified channel
         */
        public static Motor jaguar(int channel,
                                   DoubleToDoubleFunction speedLimiter) {...}

        /**
         * Create a motor driven by a Victor speed controller on the
         * specified channel. The speed output is limited to [-1.0,1.0]
         * inclusive.
         *
         * @param channel the channel the controller is connected to
         * @return a motor on the specified channel
         */
        public static Motor victor(int channel) {...}

        /**
         * Create a motor driven by a Victor speed controller on the
         * specified channel, with a custom speed limiting function.
         *
         * @param channel the channel the controller is connected to
         * @param speedLimiter function that will be used to limit
         *        the speed; may not be null
         * @return a motor on the specified channel
         */
        public static Motor victor(int channel,
                                   DoubleToDoubleFunction speedLimiter) {...}

        /**
         * Creates a TalonSRX motor controlled by a Talon SRX with
         * built-in current sensor and position (angle) sensor.
         * The WPILib's CANTalon object passed into this method should
         * be already configured by the calling code.
         *
         * @param talon the already configured WPILib CANTalon instance;
         *        may not be null
         * @param pulsesPerDegree the number of encoder pulses per
         *        degree of revolution of the final shaft
         * @return a {@link TalonSRX} motor; never null
         */
        public static TalonSRX talonSRX(CANTalon talon,
                                        double pulsesPerDegree) {...}
    }
    ...
}
```

Obviously you should only use this class in code that will only run on the robot, so we recommend centralizing this logic inside one of your top-level robot classes:

```
public class SimpleRobot extends IterativeRobot {

    private TankDrive drive;

    @Override
    public void robotInit() {
        ...
        Motor left = Motor.compose(Hardware.Motors.talon(LF_MOTOR_PORT),
                                   Hardware.Motors.talon(LR_MOTOR_PORT));
        Motor right = Motor.compose(Hardware.Motors.talon(RF_MOTOR_PORT),
                                    Hardware.Motors.talon(RR_MOTOR_PORT))
                                    .invert();
        drive = new TankDrive(left, right);
    }
```

You can then pass the motors around to other components that need them.

Tip	Testability tip You might think that it’s easy for components to just get the motors or `drive` field on the `SimpleRobot`. However, that embeds knowledge about how to find the sensors inside the component that uses it. This makes it very difficult to test the component off-robot, because it will always try to get a hardware-based sensor from the `SimpleRobot` object. Instead, it’s much better to use *injection*, which is just a fancy way of saying that when a component is created it should be handed all the objects it needs. So if we have a component that requires a sensor, then that component should require the sensor be given to it (usually through the constructor, if possible). The robot class can create this component and pass the sensor to it, while test cases can pass in a mock sensor. (We’ll talk about mocks in the next section.) The bottom line is that these other components should depend only upon *injected* `SpeedSensor`, `SpeedController`, `Motor`, `LimitedMotor`, or `TalonSRX` objects, and should never *know* how to find one. This helps keep these other components *testable*.

Using motors in tests

Your tests should never use the Hardware class to create motors in your tests. Instead, Strongback provides a series of *mock* classes that can be used in place of the hardware implementations. These mocks extend the normal interfaces as expected, but they add setter methods that your test code can explicitly set up the mock motors.

The `MockMotor` class is defined as follows:

```
public class MockMotor implements Motor {

    private volatile double speed = 0;

    MockMotor(double speed) {
        this.speed = speed;
    }

    @Override
    public double getSpeed() {
        return speed;
    }

    @Override
    public MockMotor setSpeed(double speed) {
        this.speed = speed;
        return this;
    }

    public MockMotor invert() {...}
}
```

You can create mock objects using the `org.strongback.mock.Mock` class static factory methods. For example, here's a fragment from a test that creates a few mocks motors in a stopped or running initial state:

```
MockMotor m1 = Mock.runningMotor(0.5);
MockMotor m2 = Mock.stoppedMotor();

// Pass the motors to components that use them, and verify they behave
// correctly given the two speeds

m2.setSpeed(0.4);

// Check the components are using the motors correctly
```

Strongback also provides a `MockTalonSRX` , and too these can be constructed using the `Mock` static factory methods. However, there is no mock `LimitedMotor` , since a real `LimitedMotor` can be used with a `MockMotor` object and `MockSwitches` :

```
MockSwitch forLimit = Mock.notTriggeredSwitch();
MockSwitch revLimit = Mock.notTriggeredSwitch();
MockMotor motor = Mock.runningMotor(0.5);
LimitedMotor limited = LimitedMotor.create(motor,forLimit,revLimit);

// The limited motor is moving forward at 50% speed and neither limit
// is triggered. Test a component using this limited motor ...

// Change the motor speed ..
motor.setSpeed(0.4);

// Check the component correctly uses the limited motor ...

// Stop at the limit ...
forLimit.setTriggered();

// Check the component correctly uses the limited motor ...
```

Solenoid

A hardware solenoid is any actuation device that can be extended or retracted. Examples include pneumatic cylinders and electrical solenoids.

This section describes Strongback's multiple interfaces that represent physical solenoids.

Solenoid

The `org.strongback.components.Solenoid` interface represents a simple actuator device that can extend or retract. Many solenoids will be nearly instantaneous, although others may be extending or retracting for more than one control cycle.

```
@ThreadSafe
public interface Solenoid extends Requirable {

    /**
     * The direction of the solenoid.
     */
    static enum Direction {
        /** The solenoid is extending. */
        EXTENDING,
        /** The solenoid is retracting. */
        RETRACTING,
        /** The solenoid is stopped. */
        STOPPED;
    }

    /**
     * Get the current direction of this solenoid.
     * @return the current direction; never null
     */
    Direction getDirection();

    /**
     * Extends this solenoid.
     * @return this object to allow chaining of methods; never null
     */
    Solenoid extend();

    /**
     * Retracts this solenoid.
     * @return this object to allow chaining of methods; never null
     */
    Solenoid retract();

    /**
```

```
     * Determine if this solenoid is or was extending.
     *
     * @return true if this solenoid is in the process of extending but not yet
     *     fully extended, or false otherwise.
     */
    default boolean isExtending() {...}

    /**
     * Determine if this solenoid is or was retracting.
     *
     * @return true if this solenoid is in the process of retracting but not yet
     *     fully extended, or false otherwise.
     */
    default boolean isRetracting() {...}

    /**
     * Determine if this solenoid is stopped.
     *
     * @return {@code true} if this solenoid is not retracting or extending, or
     *     false otherwise
     */
    default boolean isStopped() {...}
}
```

Most of the methods should be easy to understand and use. The `getDirection()` method returns an enumeration that represents last known direction commanded to the solenoid.

SolenoidWithPosition

The `org.strongback.components.SolenoidWithPosition` interface is a type of `Solenoid` that is able to monitor its own position. An example might be a pneumatic cylinder with magnetic reed switches.

```
@ThreadSafe
public interface SolenoidWithPosition extends Solenoid {

    /**
     * The possible positions for a limited motor.
     */
    public enum Position {
        /** The motor is fully extended. **/
        EXTENDED,
        /** The motor is fully retracted. **/
        RETRACTED,
        /** The motor is not fully retracted or fully extended, but the exact position is
        UNKNOWN
    }

    /**
```

```
         * Get the current position of this solenoid.
         *
         * @return the current position; never null
         */
        public Position getPosition();

        /**
         * Determines if this Solenoid is extended.
         *
         * @return true if this solenoid is fully extended, or false otherwise
         */
        default public boolean isExtended() {...}

        /**
         * Determines if this Solenoid is retracted.
         *
         * @return true if this solenoid is fully retracted, or false otherwise
         */
        default public boolean isRetracted() {...}

        /**
         * Create a solenoid that uses the supplied function to determine the position.
         * @param solenoid the solenoid; may not be null
         * @param positionSupplier the function that returns the position; may not be null
         * @return the {@link SolenoidWithPosition} instance; never null
         */
        public static SolenoidWithPosition create(Solenoid solenoid,
                                                  Supplier<Position> positionSupplier ) {...}

        /**
         * Create a solenoid that uses the two given switches to determine position.
         * @param solenoid the solenoid; may not be null
         * @param retractSwitch the switch that determines if the solenoid is retracted; may
         * @param extendSwitch the switch that determines if the solenoid is extended; may no
         * @return the {@link SolenoidWithPosition} instance; never null
         */
        public static SolenoidWithPosition create(Solenoid solenoid,
                                                  Switch retractSwitch,
                                                  Switch extendSwitch ) {...}
    }
```

Again, the methods are pretty straightforward.

The `create(…)` static factory methods make it easy to create a `SolenoidWithPosition` from a `Solenoid` and one or two `Switch` objects.

Relay

The `org.strongback.components.Relay` interface that represents a device that can be turned on and off. A common example of a relay is the VEX Robotics Spike style relay used in FRC with pneumatic cylinders.

Note that a relay has one of 5 possible states:

- ON - the relay is in the "on" position;
- OFF - the relay is in the "off" position;
- SWITCHING_ON - the relay was in the "off" position but has been changed and is not yet in the "on" position;
- SWITCHING_OFF - the relay was in the "on" position but has been changed and is not yet in the "off" position; and
- UNKNOWN - the relay position is not known

Not all Relay implementations use all relay states. Very simple relays that have no delay will simply only use `State.ON` and `State.OFF`, while relays that have some delay might also use `State.SWITCHING_ON` and `State.SWITCHING_OFF`. Those relay implementations that may not know their position upon startup may start out in the `UNKNOWN` state.

```
public interface Relay {

    static enum State {
        /**
         * The relay is presently switching into the "ON" state but has
         * not yet completed the change.
         */
        SWITCHING_ON,
        /**
         * The relay is presently in the "on" position.
         */
        ON,
        /**
         * The relay is presently switching into the "OFF" state but has
         * not yet completed the change.
         */
        SWITCHING_OFF,
        /**
         * The relay is presently in the "off" position.
         */
        OFF,
        /**
         * The actual state of the relay is not known.
         */
        UNKOWN
    }
    /**
     * Get the current state of this relay.
     * @return the current state; never null
     */
    State state();

    /**
     * Turn on this relay.
     *
     * @return this object to allow chaining of methods; never null
     */
    Relay on();

    /**
     * Turn off this relay.
     *
     * @return this object to allow chaining of methods; never null
     */
    Relay off();

    /**
     * Check whether this relay is known to be on. This is equivalent to calling
     * `state() == State.ON`.
     *
     * @return true if this relay is on; or false otherwise
     */
    default boolean isOn() {...}

    /**
     * Check whether this relay is known to be off. This is equivalent to calling
     * `state() == State.OFF`.
     *
     * @return true if this relay is off; or false otherwise
     */
    default boolean isOff() {...}

    /**
     * Check if this relay is switching on. This is equivalent to calling
     * `state() == State.SWITCHING_ON`.
     *
     * @return true if this relay is in the process of switching from off to on;
     * or false otherwise
     */
    default boolean isSwitchingOn() {...}

    /**
     * Check if this relay is switching off. This is equivalent to calling
     * `state() == State.SWITCHING_OFF`.
     *
     * @return true if this relay is in the process of switching from on to off;
     * or false otherwise
     */
    default boolean isSwitchingOff() {...}
    /**
     * Obtain a relay that remains in one fixed state, regardless of any
     * calls to on() or off().
     *
     * @param state the fixed state; may not be null
     * @return the constant relay; never null
     */
    static Relay fixed(State state) {...}
}
```

Conceptually a relay is similar to a `Solenoid`, although semantics are slightly different.

Obtaining hardware solenoids and relays

Strongback provides a `org.strongback.hardware.Hardware` class with static factory methods for physical hardware solenoids and relays:

```
public class Hardware {
    ...

    /**
     * Factory methods for solenoids.
     */
    public static final class Solenoids {
        /**
         * Create a double-acting solenoid that uses the specified channels on the defaul
         *
         * @param extendChannel the channel that extends the solenoid
         * @param retractChannel the channel that retracts the solenoid
         * @param initialDirection the initial direction for the solenoid; may not be nul
         * @return a solenoid on the specified channels; never null
         */
        public static Solenoid doubleSolenoid(int extendChannel,
                                              int retractChannel,
                                              Solenoid.Direction initialDirection) {...}

        /**
         * Create a double-acting solenoid that uses the specified channels on the given
         *
         * @param module the module for the channels
         * @param extendChannel the channel that extends the solenoid
         * @param retractChannel the channel that retracts the solenoid
         * @param initialDirection the initial direction for the solenoid; may not be nul
         * @return a solenoid on the specified channels; never null
         */
        public static Solenoid doubleSolenoid(int module,
                                              int extendChannel,
                                              int retractChannel,
                                              Solenoid.Direction initialDirection) {...}

        /**
         * Create a relay on the specified channel.
         *
         * @param channel the channel the relay is connected to
         * @return a relay on the specified channel
         */
        public static Relay relay(int channel) {...}
    }
    ...
}
```

Obviously you should only use this class in code that will only run on the robot, so we recommend centralizing this logic inside one of your top-level robot classes:

```
public class SimpleRobot extends IterativeRobot {

    private Solenoid cylinder;

    @Override
    public void robotInit() {
        ...
        // pressure solenoid in channels 3 and 4
        Solenoid actualCylinder = Hardare.Solenoids.doubleSolenoid(3,4);
        // reed switches in channels 5 and 6
        Switch retracted = Hardware.Switches.normallyOpen(5);
        Switch extended = Hardware.Switches.normallyOpen(5);

        cylinder = SolenoidWithPosition.create(actualCylinder,retracted,extended);

        // pass 'cylinder' to the objects that need it ...
    }
```

You can then pass the switches around to other components that need them.

Tip	Testability tip You might think that it's easy for components to just get the `cylinder` field on the `SimpleRobot` , perhaps via getter methods. However, that embeds knowledge about how to find the sensors inside the component that uses it. This makes it very difficult to test the component off-robot, because it will always try to get a hardware-based sensor from the `SimpleRobot` object. Instead, it's much better to use *injection*, which is just a fancy way of saying that when a component is created it should be handed all the objects it needs. So if we have a component that requires a sensor, then that component should require the sensor be given to it (usually through the constructor, if possible). The robot class can create this component and pass the sensor to it, while test cases can pass in a mock sensor. (We'll talk about mocks in the next section.) The bottom line is that these other components should depend only upon *injected* `Solenoid` or `SolenoidWithPosition` objects, and should never *know* how to find one. This helps keep these other components *testable*.

Using switches in tests

Your tests should never use the Hardware class to create solenoids in your tests. Instead, Strongback provides a series of *mock* classes that can be used in place of the hardware implementations. The mock solenoid extend the `Solenoid` interface as expected, and normally this will be all your tests need. However, the `MockSolenoid` class does add a `stop()` method that stops the solenoid in whatever position it is (should that be necessary).

The `MockSolenoid` class is defined as follows:

```
public class MockSolenoid implements Solenoid {

    private volatile Direction direction = Direction.STOPPED;
    private final boolean completeImmediately;

    protected MockSolenoid( boolean completeImmediately ) {
        this.completeImmediately = completeImmediately;
    }

    @Override
    public MockSolenoid extend() {
        direction = Direction.EXTENDING;
        if ( completeImmediately ) direction = Direction.STOPPED;
        return this;
    }

    @Override
    public MockSolenoid retract() {
        direction = Direction.EXTENDING;
        if ( completeImmediately ) direction = Direction.STOPPED;
        return this;
    }

    @Override
    public Direction getDirection() {
        return direction;
    }

    /**
     * Stop any movement of this solenoid.
     * @return this object so that methods can be chained together; never null
     */
    public MockSolenoid stop() {
        direction = Direction.STOPPED;
        return this;
    }
}
```

You can create mock objects using the `org.strongback.mock.Mock` class' static factory methods. For example, here's a fragment from a test that creates a few instantaneous solenoids:

```
Solenoid s1 = Mock.instantaneousSolenoid();
s1.extend();
assert s1.isExtending();
```

There is no mock `SolenoidWithPosition`; instead, just create a mock solenoid with one or two mock switches:

```
Solenoid s1 = Mock.instantaneousSolenoid();
MockSwitch extended = Mock.notTriggered();
MockSwitch retracted = Mock.triggered();
SolenoidWithPosition cylinder = SolenoidWithPosition.create(s1,retracted,extended);
cylinder.extend();
assertThat(extended.isTriggered()).isTrue();
```

The `MockRelay` class is defined as follows:

```
public class MockRelay implements Relay {

    private State state;

    @Override
    public MockRelay off() {
        state = State.OFF;
        return this;
    }

    @Override
    public MockRelay on() {
        state = State.ON;
        return this;
    }

    /**
     * Set the state of this relay to State.SWITCHING_OFF.
     * @return this instance to enable chaining methods; never null
     */
    public MockRelay switchingOff() {
        state = State.SWITCHING_OFF;
        return this;
    }

    /**
     * Set the state of this relay to State.SWITCHING_ON.
     * @return this instance to enable chaining methods; never null
     */
    public MockRelay switchingOn() {
        state = State.SWITCHING_ON;
        return this;
    }

    @Override
    public State state() {
        return state;
    }
}
```

Using the mock relay is easy, although you’ll likely do this in tests of components that use the relay.

```
MockRelay relay = Mock.relay().on();
```

Drives

Most FRC robots have a drive system that includes multiple motors that are controlled in various ways. Although there are lots of styles of chassis, several are common enough that Strongback provides components defined in terms of its hardware components, with methods that make it easy to drive in autonomous or teleoperated modes.

TankDrive

The `TankDrive` component represents chassis with tank/skid-style drive system, where the wheels on one side of the robot are controlled by one or more motors, and are independent from the wheels and motors on the other side of the robot. This class provides methods to drive using arcade-style, tank-style, and cheesy-style inputs. The drive also implements Requirable so that Commands can depend upon it directly when executing.

```
public class TankDrive implements Requirable {

    /**
     * Creates a new DriveSystem subsystem that uses the supplied drive train and
     * no shifter. The voltage send to the drive train is limited to [-1.0,1.0].
     *
     * @param left the left motor on the drive train for the robot; may not be null
     * @param right the right motor on the drive train for the robot; may not be null
     */
    public TankDrive(Motor left, Motor right) {...}

    /**
     * Creates a new DriveSystem subsystem that uses the supplied drive train and
     * optional shifter. The voltage send to the drive train is limited to [-1.0,1.0].
     *
     * @param left the left motor on the drive train for the robot; may not be null
     * @param right the right motor on the drive train for the robot; may not be null
     * @param shifter the optional shifter used to put the transmission into high gear;
     *        may be null if there is no shifter
     */
    public TankDrive(Motor left, Motor right, Relay shifter) {...}

    /**
     * Creates a new DriveSystem subsystem that uses the supplied drive train and
     * optional shifter. The voltage send to the drive train is limited by the given
     * function.
     *
     * @param left the left motor on the drive train for the robot; may not be null
     * @param right the right motor on the drive train for the robot; may not be null
     * @param shifter the optional shifter used to put the transmission into high gear;
     *        may be null
     * @param speedLimiter the function that limits the speed sent to the drive train;
     *        if null, then a default clamping function is used to limit to the range
     *        [-1.0,1.0]
     */
    public TankDrive(Motor left, Motor right, Relay shifter,
                     DoubleToDoubleFunction speedLimiter) {...}

    /**
     * Shift the transmission into high gear. This method does nothing if the
     * drive train has no transmission shifter.
     */
    public void highGear() {...}

    /**
     * Shift the transmission into low gear. This method does nothing if the
     * drive train has no transmission shifter.
     */
    public void lowGear() {...}

    /**
     * Stop the drive train. This sets the left and right motor speeds to 0,
     * and shifts to low gear.
     */
    public void stop() {...}

    /**
     * Arcade drive implements single stick driving. This function lets you directly
     * provide joystick values from any source.
     *
     * @param driveSpeed the value to use for forwards/backwards; must be -1 to 1, inclus
     * @param turnSpeed the value to use for the rotate right/left; must be -1 to 1, incl
     */
    public void arcade(double driveSpeed, double turnSpeed) {...}

    /**
     * Arcade drive implements single stick driving. This function lets you directly
     * provide joystick values from any source.
     *
     * @param driveSpeed the value to use for forwards/backwards; must be -1 to 1, inclus
     * @param turnSpeed the value to use for the rotate right/left; must be -1 to 1, incl
     *        Negative values turn right; positive values turn left.
     * @param squaredInputs if set, decreases the sensitivity at low speeds
     */
    public void arcade(double driveSpeed, double turnSpeed, boolean squaredInputs) {...}

    /**
     * Provide tank steering using the stored robot configuration. This function lets you
     * directly provide joystick values from any source.
     *
     * @param leftSpeed The value of the left stick; must be -1 to 1, inclusive
     * @param rightSpeed The value of the right stick; must be -1 to 1, inclusive
     * @param squaredInputs Setting this parameter to true decreases the sensitivity
     *        at lower speeds
     */
    public void tank(double leftSpeed, double rightSpeed, boolean squaredInputs) {...}

    /**
     * Provide tank steering using the stored robot configuration. This function
     * lets you directly provide joystick values from any source.
     *
     * @param leftSpeed The value of the left stick; must be -1 to 1, inclusive
     * @param rightSpeed The value of the right stick; must be -1 to 1, inclusive
     */
    public void tank(double leftSpeed, double rightSpeed) {...}

    /**
     * Provide "cheesy drive" steering using a steering wheel and throttle.
     * This function lets you directly provide joystick values from any source.
     *
     * @param throttle the value of the throttle; must be -1 to 1, inclusive
     * @param wheel the value of the steering wheel; must be -1 to 1, inclusive.
     *        Negative values turn right; positive values turn left.
     * @param isQuickTurn true if the quick-turn button is pressed
     */
    public void cheesy(double throttle, double wheel, boolean isQuickTurn) {...}
}
```

There are several constructors that take a left and right `Motor` , and an optional `Relay` for the shifter. By default all inputs will be limited between -1.0 and 1.0 , but a third contructor allows you to pass in a custom function that limits the values to a different range.

Many drive trains have multiple motors on each side, so simply use `Motor.compose(…)` to create a logical single motor for each side of the robot from multiple motors. Here's an example that uses two motors on each side:

```
Motor leftFront = ...
Motor leftRear = ...
Motor rightFront = ...
Motor rightRear = ...
Motor left = Motor.compose(leftFront, leftRear);
Motor right = Motor.compose(rightFront, rightRear);
TankDrive drive = new TankDrive(left, right);
```

The `TankDrive` class has `highGear()` and `lowGear()` methods that, if a `Relay` for the transmission is supplied in the constructor, shift the transmission between high and low gears. The `stop()` method stops all motors.

The `TankDrive` class can be controlled in several different ways:

1. **Arcade Style** - Uses an arcade-style joystick: moving the joystick forward makes the robot move forward at a speed proportional to the joystick position; moving it backwards makes the robot go backward proportional to the joystick position; moving the joystick to the left turns the robot to the left at a turn speed proportional to the joystick position; and moving the joystick to the right turns the robot to the right at a turn speed proportional to the joystick position.
2. **Tank Style** - Uses two joysticks (perhaps on an Xbox-style gamepad controller), where the left joystick controls the speed of the left motors, and the right joystick controls the speed of the right motors. Both joysticks are moved forward to make the robot drive forward, or moved backward to make the robot drive backward. A difference in the position of the left and right joysticks causes the robot to turn.
3. **Cheesy Drive** - Inspired by Team 254, *cheesy drive* uses a steering wheel to turn left and right, and a separate throttle to control speed. A quick-turn parameter increases the turning speed at slower forward/reverse speeds.

The following fragment shows how to drive forward at 50% speed using the *arcade* style control:

```
drive.arcade(0.5, 0.0);
```

and *tank* style control:

```
drive.tank(0.5, 0.5);
```

and *cheesy* style control:

```
drive.cheesy(0.5, 0.0, false);
```

Normally the parameters are read from user input devices. For example, here's most of a sample robot program that sets up the motors, `TankDrive` and speed ranges from a Logitech Attack 3D user input device connected to the Driver Station:

```
public class SimpleTankDriveRobot extends IterativeRobot {

    ....

    private TankDrive drive;
    private ContinuousRange driveSpeed;
    private ContinuousRange turnSpeed;

    @Override
    public void robotInit() {
        ...
        Motor left = Motor.compose(Hardware.Motors.talon(1), Hardware.Motors.talon(2));
        Motor right = Motor.compose(Hardware.Motors.talon(3), Hardware.Motors.talon(4)).i
        drive = new TankDrive(left, right);

        FlightStick joystick = Hardware.HumanInterfaceDevices.logitechAttack3D(1);
        driveSpeed = joystick.getPitch();
        turnSpeed = joystick.getRoll().invert();

        ...
    }

    @Override
    public void teleopPeriodic() {
        drive.arcade(driveSpeed.read(), turnSpeed.read());
    }
}
```

The `robotInit()` method does all of the initialization of the `TankDrive` and user input device, and it creates two `ContinuousRange` objects from the joystick pitch and roll axis. (Note that the turn speed is inverted based upon the sign of the joystick and that of the `TankDrive`.)

The `teleopPeriodic()` method is called many times each second, and it gets the drive speed and turn speed from the appropriate ranges and calls `drive.arcade(…)` with the two speeds.

Here's a very similar example that uses *tank* style inputs with an Xbox-style controller:

```
public class SimpleTankDriveRobot extends IterativeRobot {

    ....

    private TankDrive drive;
    private ContinuousRange leftSpeed;
    private ContinuousRange rightSpeed;

    @Override
    public void robotInit() {
        ...
        Motor left = Motor.compose(Hardware.Motors.talon(1), Hardware.Motors.talon(2));
        Motor right = Motor.compose(Hardware.Motors.talon(3), Hardware.Motors.talon(4)).i
        drive = new TankDrive(left, right);

        Gamepad pad = Hardware.HumanInterfaceDevices.logitechDualAction(1);
        leftSpeed = pad.getLeftY();
        rightSpeed = pad.getRightY();

        ...
    }

    @Override
    public void teleopPeriodic() {
        drive.tank(leftSpeed.read(), rightSpeed.read());
    }
}
```

Other than renaming the `ContinuousRange` fields, the bulk of the difference is in the `robotInit()` method and the setup of the two `ContinuousRange` fields.

Regardless of the style of control that is used in teleoperated mode, autonomous mode can use any of the three control methods. Simply use the methods that best match your autonomous control logic.

Command Framework

Strongback's command framework makes it easier to write robot code that *does multiple things at once*. Using the command framework is entirely optional: your robot might not need it, or if it does you might prefer to use another framework, such as the command-based framework built into WPILib.

What are commands and why are they useful? Many robots need to do multiple things at once, and Strongback's commands provide a very simple, composeable, and testable way to organize the code to control these different activities. Plus, commands can be used in both autonomous and teleoperated modes.

When you use Strongback commands, you write each separate bit of logic to control these different parts as a *command* class. You can even create larger and more complex commands, called *command groups*, by composing them from a number of smaller, more atomic commands. Then at the appropriate time, your code creates an instance of a command and submits it to the *scheduler*. The scheduler's job is to maintain a list of all submitted commands and, using a single thread, periodically go through this list and give each command an opportunity to execute. The scheduler removes a command when that command tells the scheduler it has finished completes, or if/when the scheduler receives a new command that preempts it.

The rest of this chapter starts with an example that shows why commands are useful, and then walks you through the details of how to use commands in your robot code and the flexibility and power that Strongback's command framework provides. We then cover how to test commands, and finally end by comparing Strongback's command framework to WPILib's command-based programming.

Why Commands

Before we dive into more detail about how to use the command framework work, let’s start by showing what commands are and why they’re better than the alternative.

Imagine a robot with an articulating 2-join arm that can swivel on the chassis. At the end of the arm is a claw that can open and close. The swivel and each joint has a motor and an encoder so that it knows the angle of the joint, and a pneumatic actuator opens and closes the joint.

First of all, let’s look at the Strongback components that the robot code might use:

```
Motor swivelMotor = ...
Motor shoulderMotor = ...
Motor elbowMotor = ...
Compass swivelAngle = ...
AngleSensor shoulderAngle = ...
AngleSensor elbowAngle = ...
Solenoid claw = ...
```

(We use a `Compass` for `swivelAngle` instead of an `AngleSensor` because our arm can swivel more than 360º.)

Based upon the game, the robot needs to be able to move the arm to grab a game piece, but that means swiveling the arm to 30º to the right of center, opening the claw, and raising the claw to be 25" off the ground and 12" from the front of the robot (which the mechanical design team tells you corresponds to a shoulder angle of 64º and an elbow angle of 83º). Your job as a programmer is to make this happen, regardless of how the arm is currently positioned.

Using very simplistic sequential-style logic, you might initially try something like this:

```
// First swivel the arm to the correct angle (within +/- 1 degree) ...
while ( true ) {
    double diff = swivelAngle.computeHeadingChangeTo(30.0,1);
    if ( diff < 0.0 ) {
        swivelMotor.setSpeed(-0.4);
    } else if ( diff > 0.0 ) {
        swivelMotor.setSpeed(0.4);
    } else {
       swivelMotor.stop();
       break;
    }
}

// Raise the shoulder to the correct angle ...
while ( true ) {
    double diff = shoulderAngle.getAngleChangeTo(64.0,1);
    if ( diff < 0.0 ) {
        shoulderMotor.setSpeed(-0.4);
    } else if ( shoulderAngle.getAngleChangeTo(64.0,1) > 0.0 ) {
        shoulderMotor.setSpeed(0.4);
    } else {
       shoulderMotor.stop();
       break;
    }
}

// Move the elbow to the correct angle ...
while ( true ) {
    double diff = elbowAngle.getAngleChangeTo(83.0,1);
    if ( diff < 0.0 ) {
        elbowMotor.setSpeed(-0.4);
    } else if ( diff > 0.0 ) {
        elbowMotor.setSpeed(0.4);
    } else {
       elbowMotor.stop();
       break;
    }
}

// Open the claw ...
claw.retract();
```

Now, we surely could rewrite this to make it more readable and more efficient, or we could put this logic inside a class so that we don't have to duplicate it in multiple places. Yes, we could make the code better in a number of ways. But no matter what we do, *this code will always take multiple seconds to run, or as long as needed until the hardware moves to the desired physical orientation*. And while that is happening, the robot really can't do anything else. (Well, it could, but you'd have to embed that other logic inside this block of code! Try making your robot do 5 or even 10 things at once — yuck!)

Commands make this far easier and simpler. We'll create a separate `Command` subclass for each movable part: `SwivelArm`, `RotateShoulder`, `RotateElbow`, `OpenClaw`, and `CloseClaw`. Let's start with `SwivelArm`:

```
public class SwivelArm extends Command {
    private final Motor swivel;
    private final Compass angle;
    private final double speed;
    private final double desiredAngle;
    private final double tolerance;
    public SwivelArm( Motor swivel, Compass angle, double speed,
                      double desiredAngleInDegrees, double toleranceInDegrees ) {
        super(swivel);
        this.swivel = swivel;
        this.angle = angle;
        this.speed = Math.abs(speed);
        this.desiredAngle = desiredAngleInDegrees;
        this.tolerance = toleranceInDegrees;
    }
    @Override
    public boolean execute() {
        double diff = angle.computeHeadingChangeTo(desiredAngleInDegrees,toleranceInDegre
        if ( diff == 0.0 ) {
            swivel.stop();
            return true;
        }
        if ( diff < 0.0 ) {
            swivel.setSpeed(-speed);
        } else {
            swivel.setSpeed(speed);
        }
        return false;
    }
}
```

The `RotateShoulder` and `RotateElbow` commands will actually be nearly identical. Rather than copy all that code, let's refactor `SwivelArm` to put most of the logic inside a base class. And because one would deal with a `Compass` and the other two a `AngleSensor`, let's actually use Java 8 lambdas and `java.util.function.DoubleSupplier` to tell us whether we need to move the motor. The result is a `MoveDevice` class that we can use for *any* motor:

```
public class MoveDevice extends Command {
    private final Motor swivel;
    private final DoubleSupplier moveDirection;
    private final double speed;
    public SwivelArm( Motor swivel, double speed, DoubleSupplier moveDirection ) {
        super(swivel);
        this.swivel = swivel;
        this.speed = Math.abs(speed);
        this.moveDirection = moveDirection;
    }
    @Override
    public boolean execute() {
        double diff = moveDirection.getAsDouble();
        if ( diff == 0.0 ) {
            swivel.stop();
            return true;
        }
        if ( diff < 0.0 ) {
            swivel.setSpeed(-speed);
        } else {
            swivel.setSpeed(speed);
        }
        return false;
    }
}

public class SwivelArm extends MoveDevice {
    public SwivelArm( Motor motor, Compass angle, double speed,
                      double desiredHeadingInDegrees, double toleranceInDegrees ) {
        super(motor,speed,()->angle.computeHeadingChangeTo(desiredHeadingInDegrees,tolera
    }
}

public class RotateShoulder extends MoveDevice {
    public RotateShoulder( Motor motor, Angle angle, double speed,
                           double desiredAngleInDegrees, double toleranceInDegrees ) {
        super(motor,speed,()->angle.computeAngleChangeTo(desiredAngleInDegrees,toleranceI
    }
}

public class RotateElbow extends MoveDevice {
    public RotateElbow( Motor motor, Angle angle, double speed,
                        double desiredAngleInDegrees, double toleranceInDegrees ) {
        super(motor,speed,()->angle.computeAngleChangeTo(desiredAngleInDegrees,toleranceI
    }
}

public class SolenoidOperation extends Command {
    public SolenoidOperation( Solenoid solenoid ) {
    }
}

public class OpenClaw extends Command {
    private final Solenoid solenoid;
    public OpenClaw( Solenoid solenoid ) {
        super(solenoid);
        this.solenoid = solenoid;
    }
    @Override
    public boolean execute() {
        this.solenoid.retract();
        return true;
    }
}

public class CloseClaw extends SolenoidOperation {
    private final Solenoid solenoid;
    public CloseClaw( Solenoid solenoid ) {
        super(solenoid);
        this.solenoid = solenoid;
    }
    @Override
    public boolean execute() {
        this.solenoid.extend();
        return true;
    }
}
```

We can then make our robot position the arm and claw with this simple code:

```
Strongback.submit(new SwivelArm(swivelMotor,swivelAngle,0.4,30.0,1.0));
Strongback.submit(new RotateShoulder(shoulderMotor,shoulderAngle,0.4,64.0,1.0));
Strongback.submit(new RotateElbow(elbowMotor,elbowAngle,0.4,83.0,1.0));
Strongback.submit(new OpenClaw(claw));
```

This is *far* simpler, and better yet all of these commands operate asynchronously (in another thread), will complete whenever they are done, and do not prevent our code from doing other things and submitting other commands! And if we submit any commands that use the arm, shoulder, elbow, or claw *while these commands are still running*, Strongback will automatically cancel any earlier commands that have the same requirements.

Now, if we find that we're regularly submitting these same commands together, we might want to create a *command group*. We could create a class, but it might be simpler and make even more readable code if we do all of this in a function that creates an anonymous class. In fact, let's create a `RobotArm` class that holds onto the components that make up the arm:

```
public class RobotArm {

    private final Motor swivel;
    private final Motor shoulder;
    private final Motor elbow;
    private final Compass swivelAngle;
    private final AngleSensor shoulderAngle;
    private final AngleSensor elbowAngle;
    private final Solenoid claw;

    public RobotArm( Motor swivel, Compass swivelAngle,
                     Motor shoulder, AngleSensor shoulderAngle,
                     Motor elbow, AngleSensor elbowAngle,
                     Solenoid claw ) {
        this.swivel = swivel;
        this.shoulder = shoulder;
        this.elbow = elbow;
        this.claw = claw;
        this.swivelAngle = swivelAngle;
        this.shoulderAngle = shoulderAngle;
        this.elbowAngle = elbowAngle;
    }

    public CommandGroup positionArm( double swivelTargetAngle,
                             double shoulderTargetAngle,
                             double elbowTargetAngle,
                             double tolerance,
                             double speed,
                             boolean openClaw ) {
        return CommandGroup.runSimultaneously(
                new SwivelArm(swivel,swivelAngle,speed,swivelTargetAngle,tolerance));
                new RotateShoulder(shoulder,shoulderAngle,speed,shoulderTargetAngle,toler
                new RotateElbow(elbow,elbowAngle,speed,elbowTargetAngle,tolerance));
                openClaw ? new OpenClaw(claw) : new CloseClaw(claw));
            );
    }

    ...
}
```

Then, to position the arm at any time, simply invoke the `positionArm` method on the `RobotArm` intance (which would be easy to do via a button):

```
Strongback.submit( robotArm.positionArm(30.0,64.0,83.0,1.0,0.4,true) );
```

Of course, you enhance the `RobotArm` class with the logic to convert desired physical positions of the claw (e.g., 30º to the right of center, claw 25" off the ground and 12" from the front of the robot) into the correct target angles:

```
public class RobotArm {
    ...
    public CommandGroup physicallyPositionArm( double swivelTargetAngle,
                                               double clawHeight,
                                               double clawDistanceForward,
                                               double speed,
                                               boolean openClaw ) {
        ...
    }
}
```

Not only would this encapsulate the arm-related components, but it can also encode your mathematical models of the physical system in a class that is *very* easy to test.

Using Commands

The previous section showed how your robot code can do more by using commands, and how this makes the code simpler, testable, and makes the logic in the commands (re)usable in both autonomous and teleoperated modes. This section goes step by step into *how* you use the command framework.

Creating Command subclasses

We saw in the earlier example how easy it is to create custom commands by writing a subclass for each command. Most of the time, each command subclass will need a constructor and an `execute():boolean` method. Here's an example we saw earlier:

```
public class OpenClaw extends Command {
    private final Solenoid solenoid;
    public OpenClaw( Solenoid solenoid ) {
        super(solenoid);
        this.solenoid = solenoid;
    }
    @Override
    public boolean execute() {
        this.solenoid.retract();
        return true;
    }
}
```

Let's look at this class in a bit more detail. Why does this class require the `Solenoid` object to be passed into the constructor? Couldn't the command just get the `Solenoid` object from our robot's top-level class? It *could* do that, but passing in the `Solenoid` makes this command *testable*. For example, here's part of a sample unit test (using JUnit and Fest assertions):

```
public class OpenClawTest {
    @Test
    public void shouldRetractSolenoidWhenSolenoidIsInitiallyExtended() {
        MockSolenoid solenoid = Mock.instantaneousSolenoid().extend();
        OpenClaw command = new OpenClaw(solenoid);
        assertThat(command.execute()).isTrue();
        assertThat(solenoid.isRetracted()).isTrue();
        assertThat(solenoid.isExtended()).isFalse();
    }
    @Test
    public void shouldRetractSolenoidWhenSolenoidIsAlreadyRetracted() {
        MockSolenoid solenoid = Mock.instantaneousSolenoid().retract();
        OpenClaw command = new OpenClaw(solenoid);
        assertThat(command.execute()).isTrue();
        assertThat(solenoid.isRetracted()).isTrue();
        assertThat(solenoid.isExtended()).isFalse();
    }
}
```

Second, our `OpenClaw` class is actually immutable, which means it has no state that changes as it goes through its lifecycle. This simplifies our code, and interestingly means you can *submit* the same `OpenClaw` *instance* multiple times (if that turns out to be an advantage for you).

What our `OpenClaw` class does not do is override other methods. One method that your subclasses can override is the `initialize()` method, which is called by the scheduler *exactly one time* right before it calls `execute()` the first time. The `initialize()` method is a great place to do any one-time setup logic (far better than having an `if` -block in your `execute()` method). By default, `initialize()` does nothing.

Another method you can override is the `end()` method, which is always called after `execute()` returns `true` . However, `end()` is not called if the command is interrupted.

The last method you can override is the `interrupted()` method, which is a signal to you that the command was interrupted before `initialize()` was called or before `execute()` could return `true` . A command might be interrupted because the command was canceled, because the robot was shutdown while the command was running, or because `initialize()` or `execute()` threw an exception. Note that if `interrupted()` is called, then `end()` will not be called on the command.

One thing you may note is that `Command` has no methods that make it run. You can create as many instances of a command as you want, and none of them will do anything until you *submit the command objects*, which we'll cover in a later section.

So, writing `Command` subclasses is very easy and straightforward, and they're very easy to test. But you don't always need to write a subclass for a custom command.

Using CommandGroup

A *command group* is a special type of `Command` that is composed of a number of other `Command` objects executed in sequence and/or in parallel. Generally, you define a command group by writing a class that extends `org.strongback.command.CommandGroup`, and specifying in the subclass' constructor which commands should be executed sequential or simultaneously. The commands passed into these methods can be any `Command` subclass, `CommandGroup` subclass, or group of sequential, simultaneous, or forked commands.

Commands executed one after the other are *sequential*, and are defined within the constructor with a call to the `sequentially(…)` method that takes an ordered array of `Command` instances. The order of the `Command` objects is important: when the `CommandGroup` is executed, it will run the first `Command` object until it completes, and only then does it run the second `Command` object until it completes, and so on. The whole `CommandGroup` finishes only after the last Command is completed. Here's a simple sequential command group that when executed, first executes an instance of `CommandA`, then an instance of `CommandB`, and finally an instance of `CommandC`:

```
public class MySequentialCommand extends CommandGroup {
    public MySequentialCommand() {
        sequentially(new CommandA(),
                     new CommandB(),
                     new CommandC());
    }
}
```

Commands executed at the same time (in parallel) are *simultaneous*, and are defined within the constructor with a call to the `simultaneously(…)` method that takes an array of `Command` instances. Order is not important, since they are all are executed concurrently. The whole `CommandGroup` finishes only when *all* `Command` objects have completed. Here's a simple simultaneous command group that when executed, executes in parallel an instance of `CommandA` and an instance of `CommandB`:

```
public class MySimultaneousCommand extends CommandGroup {
    public MySimultaneousCommand() {
        simultaneously(new CommandA(),
                       new CommandB());
    }
}
```

You can also create a `CommandGroup` that is a combination of sequential and simultaneous commands. The following code shows a complete `CommandGroup` subclass that, when executed, executes at the same time two commands (an instance of `CommandA` and an instance of `CommandB`) and after both are finished a third command object of type `CommandC` :

```
public class MyMixedCommands extends CommandGroup {
    public MyMixedCommands() {
        sequentially(
            simultaneously(new CommandA(),
                           new CommandB()),
            new CommandC());
    }
}
```

The MyMixedCommands will complete when the last of the three commands (that is, the command of type `CommandC`) completes.

It is possible for a `CommandGroup` to run one of its commands completely independently of the command group. This is called *forking*, and is defined within the constructor with a call to the `fork(…)` method that takes a single `Command` , `CommandGroup` , or the result of `sequentially(…)` or `simultaneously(…)` . Here is a more complex `CommandGroup` subclass that, when executed, first executes `CommandA` , then executes `CommandB` , then forks off `CommandC` and immediately executes `CommandD` , and finally executes both `CommandE` and `CommandF` in parallel:

```
public class MyForkCommands extends CommandGroup {
    public MyForkCommands() {
        sequentially(new CommandA(),
                     new CommandB(),
                     fork(new Command C()),
                     new CommandD());
                     new simultaneously( new CommandE(), new CommandF()));
    }
}
```

The MyForkCommands will complete when the last of the two final commands, either `CommandE` or `CommandF` , completes. Note that this is true even if `CommandC` is still running.

When creating command groups, you'll often want to pause a fixed amount of time between sequential commands. This is very easy to do with the `Command.pause(…)` method, which creates a command that is completely only after the time delay has elapsed. Here's an example that, when executed, first executes an instance of `CommandA` , then an instance of `CommandB` , then pauses for 3 seconds, and finally an instance of `CommandC` :

```
public class MySequentialCommand extends CommandGroup {
    public MySequentialCommand() {
        sequentially(new CommandA(),
                     new CommandB(),
                     Command.pause(3,TimeUnit.SECONDS)
                     new CommandC());
    }
}
```

Creating commands and command groups without subclasses

When we were designing the Strongback command framework, we wanted to make it as easy as possible to create commands and command subclasses. Of course, the easiest way to do this is to offer ways of creating custom commands without having to write a subclass. Strongback provides several ways to create commands this way.

The easiest is to create a command that runs a lambda only once:

```
Command oneTimeCommand = Command.create(()->{
   // do something
});
```

Note that the lambda doesn't return anything, making it equivalent to the `Runnable.run()` method.

You can also create a command that runs a function multiple times until a specific amount of time has elapsed. In this case, just specify the duration in seconds as a first parameter:

```
Command fixedDurationCommand = Command.create(1.5, ()->{
   // do something
});
```

If your lambda *does* return a `boolean` value that states whether the command is completed, then Strongback will repeatedly call the lambda until it returns `true` :

```
Command untileCompleteCommand = Command.create(()->{
   // do something, and determine whether it is complete ...
   boolean complete = ...
   return complete;
});
```

And of course it's very easy to create command that runs lambda until either it completes or until a maximum amount of time has elapsed is also very easy:

```
double maxDurationInSeconds = 3.0;
Command untileCompleteCommand = Command.create(maxDurationInSeconds, ()->{
   // do something, and determine whether it is complete ...
   boolean complete = ...
   return complete;
});
```

You can even create a command that cancels any currently-running command with the same requirements. This is a great way to implement a `kill switch` for a particular system. For example, maybe we'd like to have a button stop our arm from moving, wherever it is. Remember our `RobotArm` class we used earlier? We could easily add a method to that to stop the arm from swiveling by creating a new command that requires the same `Requirable` swivel motor:

```
public class RobotArm {
    ...
    public Command stopSwivelNow() {
        return Command.cancel(swivel);
    }
    ...
}
```

Of course, that method only creates the command. To run it, we have to *submit* it, and we'll learn more about that in a later section.

Finally, you can create command groups without writing a subclass using the static factory methods on `CommandGroup`. This is not quite as flexible for more complex groups, but for some simple groups it can be less code. Here's a simple sequential command group that when executed, first executes an instance of `CommandA`, then an instance of `CommandB`, and finally an instance of `CommandC`:

```
Command group = CommandGroup.sequentially(new CommandA(),
                                          new CommandB(),
                                          new CommandC());
```

and another example that creates a simultaneous command group that when executed, executes in parallel an instance of `CommandA` and an instance of `CommandB`:

```
Command group = CommandGroup.simultaneously(new CommandA(),
                                            new CommandB());
```

As you can see, these make simple command groups very concise and easy to read. Unfortunately, more complicated command groups are probably more easily coded using an explicit subclass.

Submitting Commands

Creating an instance of a `Command` does not cause it to run. To do that, you have to *submit* the command to a *scheduler*. The scheduler:

- keeps track of all submitted and currently-executing commands and command groups;
- cancels any currently-executing commands that have the same requirements as a newly-submitted command; and
- execute the current commands until they complete (or are canceled).

The `Strongback` class has its own static instance of a scheduler, and it manages it for you and ensures the scheduler operates at a very consistent frequency. By default, Strongback runs the scheduler every 5 milliseconds, although you can easily configure Strongback to use a different period. To use it, simply use the `Strongback.submit(Command)` static method we used in our earlier example.

Note	How does Strongback ensure the scheduler operates at a consistent frequency? The built-in scheduler is run using Strongback's executor, which runs all asynchronous operations using a single dedicated thread that very carefully monitors execution intervals (without using `Thread.sleep()` or scheduled threads!). Since the RoboRIO has a dual-core processor, it can run two threads without having to manage them or change thread contexts. Your main robot code runs on one thread, and Strongback's executor runs a second for all of Strongback's asynchronous operations. Strongback can also use a clock that relies upon the RoboRIO's FPGA hardware chip for more accuracy.

While most teams will find it very easy and convenient to use the built-in scheduler, a few more advanced teams may want more control over how and what executes the scheduler. If this describes you, then you can create and use your own instance of the `org.strongback.command.Scheduler`. Of course, you'll have to make sure to periodically call its `execute(long)` method when you want to execute the currently-submitted commands.

Gotchas

There are a few things to be concerned about when using commands. The first is that you must be careful not to give the scheduler too much work at the same time. Recall that the scheduler uses a separate thread to periodically execute all currently-running commands, and all commands must execute within that period. For example, if the scheduler runs every 5 milliseconds, then the total time for *all* of the commands' `execute()` calls must be less than 5 milliseconds; if not, then the scheduler will not be able to run every 5 milliseconds.

Should this happen, `Strongback` increments an internal counter that you can access via `Strongback.excessiveExecutionTimeCounts()`, and it by default logs an error message. So if you see this error message beginning with “Unable to execute all activities within … milliseconds”, then either increase the period or reduce the amount of work being concurrently executed.

Tip	If you want more control when the execution time exceeds the configured period, when you configure Strongback you can provide custom logic. Strongback will then call this function (rather than log an error message) every time the execution time is excessive.

In general, though, this won’t be a problem when commands' `execute()` methods run very quickly.

Testing commands

Strongback's command framework was designed to make testing easy. For some stateless command classes that just provide an `execute()` method, it's easy to test the basic logic with a simple unit test. We saw an example of this in an earlier section:

```
public class OpenClawTest {
    @Test
    public void shouldRetractSolenoidWhenSolenoidIsInitiallyExtended() {
        MockSolenoid solenoid = Mock.instantaneousSolenoid().extend();
        OpenClaw command = new OpenClaw(solenoid);
        assertThat(command.execute()).isTrue();
        assertThat(solenoid.isRetracted()).isTrue();
        assertThat(solenoid.isExtended()).isFalse();
    }
    @Test
    public void shouldRetractSolenoidWhenSolenoidIsAlreadyRetracted() {
        MockSolenoid solenoid = Mock.instantaneousSolenoid().retract();
        OpenClaw command = new OpenClaw(solenoid);
        assertThat(command.execute()).isTrue();
        assertThat(solenoid.isRetracted()).isTrue();
        assertThat(solenoid.isExtended()).isFalse();
    }
}
```

Other command subclasses might be more difficult to test this way. For example, if the command class has state or relies upon the `initialize()` , `interrupted()` , and/or `end()` methods, then these simple unit tests need to replicate the proper lifecycle operations on the command. To make this easier, Strongback's testing library (which your tests can depend upon but which you would not deploy to your robot) includes a `CommandTest` class that lets your test step through the various lifecycle states.

Here's an example unit test that uses `CommandTester` to test the command created by `Command.pause(…)` . It uses the time facility within `CommandTester` to manipulate the clock (independently of the JVM's clock).

```
public class PauseTest {
    @Test
    public void shouldFinishAfterTime() {
        Command command = Command.pause(5,TimeUnit.SECONDS);
        CommandTester runner = new CommandTester(command);
        assertThat(command.step(1000)).isEqualTo(false);
        assertThat(command.step(5000)).isEqualTo(false);
        assertThat(command.step(5010)).isEqualTo(true);
    }
}
```

Comparison with WPILib's Commands

Strongback's command framework is very similar to WPILib command-based framework. Indeed, if you've used the WPILib commands in the past, you probably find Strongback's framework to be easy to grasp — and probably simpler to use.

But while Strongback's command framework might be similar to that in WPILib, there are definitely some important differences.

Command classes

Both frameworks' concept of a command are very similar. The differences are in the details of how they're implemented and what this means for your FRC team.

WPILib Commands

When you use WPILib commands, you write a class for each command, and this class extends the `edu.wpi.first.wpilibj.command.Command` abstract base class. WPILib's design *requires* your class do several things:

- Provide a constructor that calls the superclass' constructor and that calls `doesRequire(…)` relay required `Subsystem` objects.
- Provide an `initialize()` method that the framework will call prior to `execute()`, and the method can perform any one-time initialization logic.
- Provide an `execute()` method that contains the logic of your command. The method doesn't return anything, and may be called multiple times depending upon the `isFinished()` method.
- Provide an `isFinished()` method that tells the framework when the command is complete.
- Provide an `end()` method that the framework will call when the command successfully finished, allowing your command to respond if needed.
- Provide an `interrupted()` method that the framework will call to let your command know it's been interrupted.

Additionally, the `edu.wpi.first.wpilibj.command.Command` base class has a few methods you are expected to use (e.g., `doesRequire(…)`) and numerous methods that you can (but shouldn't) override or call. Most of those methods deal with the command's lifecycle and state required by WPILib to properly run the command. It even has methods to `start()` and `cancel()` the command.

The last two methods make each command *know* how to execute itself, and they are a big reason why it is very difficult to unit test your command subclasses. The `edu.wpi.first.wpilibj.command.Command` is so chocked full of things that your tests simply cannot instantiate them withouth staring a good portion of the WPILib system, including those that ultimately require hardware.

Strongback Commands

When we designed Strongback, we took what we learned about using and trying to test WPILib commands and thought we could do better. We wanted to make it easy as possible to write subclasses and, where possible in some simple cases, to not even require custom subclasses. To write a custom command class with Strongback, you must:

- Provide a constructor that calls the superclass' constructor with requirements and timeout information (if there are any), and optionally call `setNotInterruptible()` if desired (all commands are interruptible by default).
- Provide an `execute()` method that contains the logic of your command and that returns whether the command is completed.

That’s it! Of course, if you want you can override any of the default methods:

- Optionally override the `initialize()` method that the framework will call prior to `execute()`, and the method can perform any one-time initialization logic. The default method does nothing, so you only have to override this if you want different behavior.
- Optionally override the `end()` method that the framework will call when the command successfully finished, allowing your command to respond if needed. The default method does nothing, so you only have to override this if you want different behavior.
- Optionally override the `interrupted()` method that the framework will call to let your command know it’s been interrupted. The default method does nothing, so you only have to override this if you want different behavior.

The Strongback `Command` base class has very few methods, no lifecycle state, and does not know how to start or run. Instead, your robot code is responsible for submitting the command for execution, typically via `Strongback.submit(Command)`.

This design makes it very easy to write *and test* custom command subclasses. For example, here is a complete `Command` subclass (minus the import statements and JavaDoc):

```
public class OpenClaw extends Command {
    private final Solenoid solenoid;
    public OpenClaw( Solenoid solenoid ) {
        super(solenoid);
        this.solenoid = solenoid;
    }
    @Override
    public boolean execute() {
        this.solenoid.retract();
        return true;
    }
}
```

Strongback was designed to use Java 8 features, so you might not even need to write a custom command subclass for all of your commands. If they're simple enough, you might be able to just pass a lambda expression to one of the `Command` class' static methods. For example, here is a simpler (albeit more anonymous and less reusable) way to define our previous `OpenClaw` command:

```
Solenoid solenoid = ...
Command openClaw = Command.create(()->solenoid.retract());
```

Other `Command` class' static methods are available for creating commands that cancel, that pause for a fixed amount of time, or than run lambdas once or multiple times and possibly for a maximum duration.

Requirements

Both frameworks have similar notions of a command *requiring* some other object(s). And in both frameworks when a command is run, the framework will cancel any previously-started command that required the same object(s).

WPILib Subsystem

With WPILib, commands could state they require one or more `edu.wpi.first.wpilibj.command.Subsystem` objects. Examples of a subsystem might be a drive train, a claw, or motors on an arm. The WPILib documentation states that all motors should be a part of a subsystem, so you need to write one `Subsystem` subclass for each of your logical subsystems.

Unfortunately the `Subsystem` class is not terribly small. It actually has all the state about which command is currently being executed, and it includes methods for communicating with the Network Tables. Despite this, it's not terribly difficult to write or use `Subsystem` subclasses. Testing them is more challenging, but that’s mostly because the motors that subsystems are to contain require hardware, and thus aren’t suitable for testing off-robot with simple unit tests.

Strongback 'Requirable'

Strongback, on the other hand, uses a far simpler notion of required objects: ß

, here are a couple of ways to create commands you can

The easiest way to do that is to offer methods that you can sometimes use you can use so that you sometimes s ways where you don’t have to actually

- **Creating command classes** - We tried to make it as easy as possible to create subclasses. Where WPILib has separate methods for `execute():void` and `isFinished():boolean`, Strongback instead combines them into a single `execute():boolean` method. Strongback also provides static methods that create commands that pause, commands that run a function one time, commands that cancel any running commands with the same requirements, commands that run a lambda, and commands that execute a function for a maximum period of time.
- **Requirements** - Whereas Strongback commands can require any object that implements the `Requirable` marker interface, WPILib commands can only depend upon `edu.wpi.first.wpilibj.command.Subsystem` subclasses that are heavyweight. Strongback behaves the same was as WPILib, but all logic is embedded in classes you don’t have to extend or subclass.
- **Running** - WPILib ryou call `run()` on the command, but ghis makes it too difficult

for common activities, like pausing a

`, and to also be able to The W`edu.wpi.first.wpilibj.command.Command` class requires your subclass override a number of methods for most commands, whereas with Strongback we tried to

Table 1. Windtrainer workouts

Strongback Class	WPILib Class	Note
org.strongback.command.Command	edu.wpi.first.wpilibj.command.Command	
	22-Aug-08	10:24
157	Worked out MSHR (max sustainable heart rate) by going hard for this interval.	22-Au 08
23:03	152	Back- back with previc interv
24-Aug-08	40:00	145

If you’ve not used either, we think you’ll find it pretty eas But we found the WPILib version to be difficult to use and very difficult to test. `Command` and `CommandGroup`

Switch Reactor

It is very common in FRC robot code to register some function that should be called when some switch or button changes state. Strongback's *switch reactor* framework makes this very easy and lightweight.

For example, the following code fragment shows how to use the reactor to submit a command whenever a Gamepad's left trigger button is pressed, and submit a different command when the same trigger is released:

```
SwitchReactor reactor = Strongback.switchReactor();
Gamepad gamepad = ...
reactor.onTriggered(gamepad.getRightTrigger(),()->Strongback.submit(new FireRepeatedlyCom
reactor.onUnTriggered(gamepad.getRightTrigger(),()->Strongback.submit(new StopFireCommand
```

When the Gamepad's right trigger is pressed, the lambda on line 3 will create and submit a `FireRepeatedlyCommand` . When the Gamepad's right trigger is released, the lambda on line 4 will create and submit a `StopFireCommand` .

Submitting a new command when a switch becomes triggered or untriggered is common enough that Strongback provides methods that make this a little easier. The following code is functionally identical to the previous sample, but is a little easier to read:

```
SwitchReactor reactor = Strongback.switchReactor();
Gamepad gamepad = ...
reactor.onTriggeredSubmit(gamepad.getRightTrigger(),FireRepeatedlyCommand::new);
reactor.onUnTriggeredSubmit(gamepad.getRightTrigger(),StopFireCommand::new);
```

Of course, you don't always want to submit a command. For example, your robot might have a button that changes the drive system gears from low to high gear. The `TankDrive` component's `highGear()` method switches to high gear, and `lowGear()` switches to low gear. We can use method references and register these methods to respond when the joystick's trigger is pressed and held or released:

```
SwitchReactor reactor = Strongback.switchReactor();
FlightStick joystick = ...
TankDrive chassis = ...
reactor.whileTriggered(joystick.getTrigger(),chassis::highGear);
reactor.whileUnTriggered(gamepad.getRightTrigger(),chassis:lowGear);
```

These functions are called only when the switch state *changes*, so pressing and holding the joystick trigger will result in one call to `TankDrive.highGear()`, and then releasing will result in a single call to `TankDrive.lowGear()`.

As you can see from the examples, the switch reactor can be accessed using the `Strongback.switchReactor()` method. The `org.strongback.SwitchReactor` interface returned by that method is quite simple:

```java
@ThreadSafe
public interface SwitchReactor {

    /**
     * Submit a {@link Command} the moment when the specified {@link Switch} is triggered
     *
     * @param swtch the {@link Switch}
     * @param commandSupplier the supplier of the command to submit; may not be null but
     * may return a null command
     */
    public void onTriggeredSubmit(Switch swtch, Supplier<Command> commandSupplier);

    /**
     * Submit a {@link Command} the moment when the specified {@link Switch} is untrigger
     *
     * @param swtch the {@link Switch}
     * @param commandSupplier the supplier of the command to submit; may not be null but
     * may return a null command
     */
    public void onUntriggeredSubmit(Switch swtch, Supplier<Command> commandSupplier);

    /**
     * Register a {@link Runnable} function that is to be called the moment
     * when the specified {@link Switch} is triggered.
     *
     * @param swtch the {@link Switch}
     * @param function the function to execute when the switch is triggered
     */
    public void onTriggered(Switch swtch, Runnable function);

    /**
     * Register a {@link Runnable} function that is to be called the moment
     * when the specified {@link Switch} is untriggered.
     *
     * @param swtch the {@link Switch}
     * @param function the function to execute when the switch is untriggered
     */
    public void onUntriggered(Switch swtch, Runnable function);

    /**
     * Register a {@link Runnable} function to be called repeatedly whenever
     * the specified {@link Switch} is in the triggered state.
     *
```

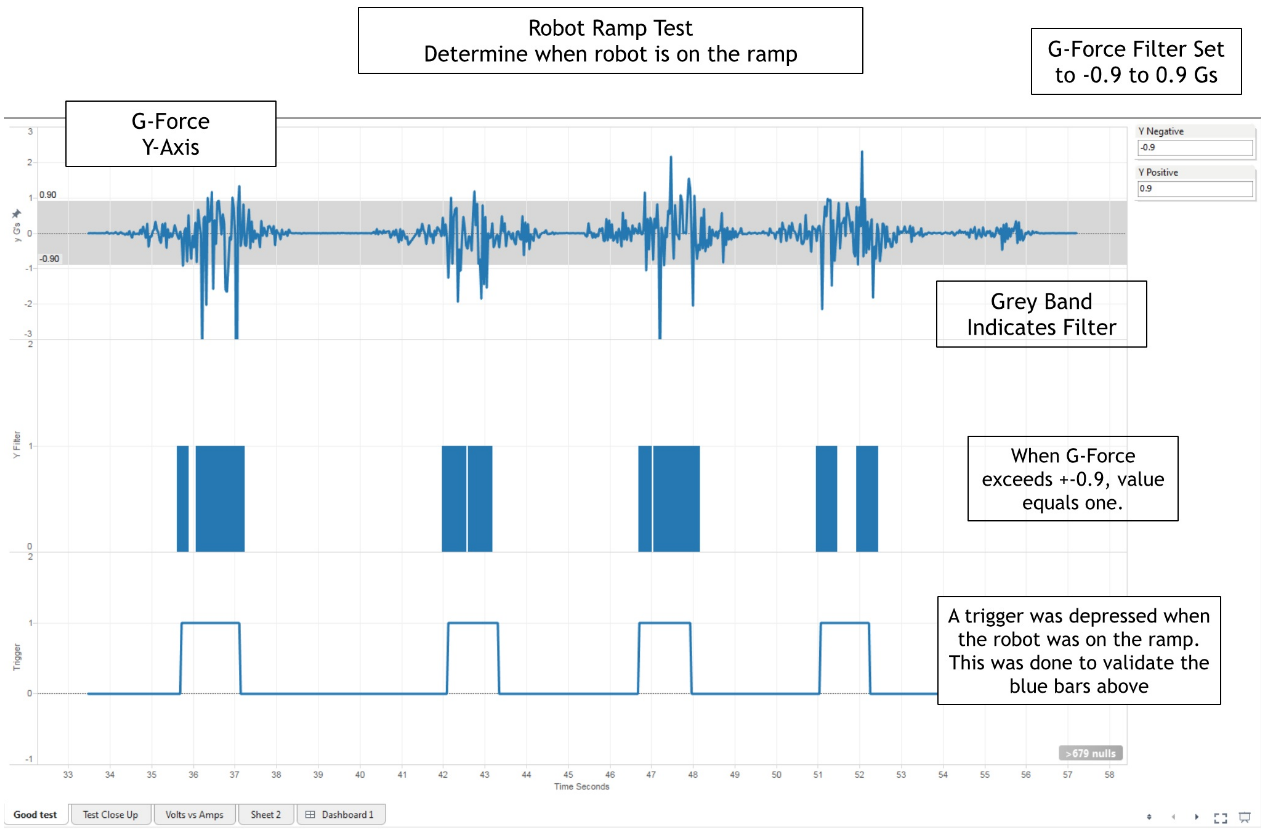
```
         * @param swtch the {@link Switch}
         * @param function the function to execute while the switch remains triggered
         */
        public void whileTriggered(Switch swtch, Runnable function);

        /**
         * Register a {@link Runnable} function to be called repeatedly whenever
         * the specified {@link Switch} is not in the triggered state.
         *
         * @param swtch the {@link Switch}
         * @param function the function to execute while the switch remains untriggered
         */
        public void whileUntriggered(Switch swtch, Runnable function);
    }
```

At this time, once a function is registered it will continue to operate.

Data and Event Recorders

Strongback lets you record various kinds of data to a log file on the RoboRIO. You can then post-process that file and extract time histories of multiple channels and events, and use tools like Excel or Tableau to visualize those histories.

The following is a graph produced with Tableau that shows on the top the y-axis acceleration (in g's), in the middle that periods of time that the acceleration exceeded 0.9 g's, and at the bottom the state of a manual button.



Strongback supports recording two kinds of data: events and data. Data are essentially continuous values, and Stronback's `DataRecorder` is used to capture at regular time steps values for all registered continuous channels. Events, on the other hand, are far less frequent occurrences of some type of action; recording them as a channel is inefficient (since there is no value at most time steps) and often makes little sense, so instead the `EventRecorder` allows components to record these spurious and infrequent events. Both the data and event records can then be combined to view a complete timeline with all available information.

Strongback's command scheduler can be configured to automatically record the state transitions of Commands as they are executed. Of course, custom robot code can also record any other kinds of events.

DataRecorder

Strongback's data recorder takes measurements every cycle of the executor, so its job is to frequently take measurements and record them. The `org.strongback.DataRecorder` interface is used to register the functions that supply the measurable values:

```
public interface DataRecorder {
    /**
     * Registers by name a function that will be periodically polled to obtain
     * and record an integer value. This method will remove any previously-registered
     * supplier, switch, or motor with the same name.
     *
     * @param name the name of this data supplier
     * @param supplier the {@link IntSupplier} of the value to be logged
     * @return this instance so methods can be chained together; never null
     * @throws IllegalArgumentException if the {@code supplier} parameter is null
     */
    public DataRecorder register(String name, IntSupplier supplier);

    /**
     * Registers by name a function that will be periodically polled to obtain
     * a double value and scale it to an integer value that can be recorded.
     * This method will remove any previously-registered supplier, switch, or
     * motor with the same name.
     *
     * @param name the name of this data supplier
     * @param scale the scale factor to multiply the supplier's double values
     *        before casting to an integer value
     * @param supplier the {@link IntSupplier} of the value to be logged
     * @return this instance so methods can be chained together; never null
     * @throws IllegalArgumentException if the {@code supplier} parameter is null
     */
    default public DataRecorder register(String name, double scale,
                                         DoubleSupplier supplier) {...}

    /**
     * Registers by name a switch that will be periodically polled to obtain
     * and record the switch state. This method will remove any previously-registered
     * supplier, switch, or motor with the same name.
     *
     * @param name the name of the {@link Switch}
     * @param swtch the {@link Switch} to be logged
     * @return this instance so methods can be chained together; never null
     * @throws IllegalArgumentException if the {@code swtch} parameter is null
     */
    public DataRecorder register(String name, Switch swtch);
    /**
     * Registers by name a speed sensor that will be periodically polled to obtain
     * and record the current speed. This method will remove any previously-registered
     * supplier, switch, or motor with the same name.
     *
     * @param name the name of the {@link SpeedSensor}
     * @param sensor the {@link SpeedSensor} to be logged
     * @return this instance so methods can be chained together; never null
     * @throws IllegalArgumentException if the {@code sensor} parameter is null
     */
    public DataRecorder register(String name, SpeedSensor sensor);
}
```

Strongback comes with a `DataRecorder` instance that is accessed via a static method:

```
DataRecorder recorder = Strongback.dataRecorder();
```

This is always done within the `robotInit()` method, before Strongback is actually started. But notice how all of the `register(…)` methods return the `DataRecorder` ; its more ideomatic to chain multiple `register(…)` methods together. Here's a sample robot with a few sensors and components, and how they are used to record their states:

```
public class SimpleTankDriveRobot extends IterativeRobot {

    ....

    private TankDrive drive;
    private ContinuousRange driveSpeed;
    private ContinuousRange turnSpeed;

    @Override
    public void robotInit() {
        ...
        Motor left = Motor.compose(Hardware.Motors.talon(1), Hardware.Motors.talon(2));
        Motor right = Motor.compose(Hardware.Motors.talon(3), Hardware.Motors.talon(4)).i
        drive = new TankDrive(left, right);

        // Set up the human input controls for teleoperated mode. We want to use the
        // Logitech Attack 3D's throttle as a "sensitivity" input to scale the drive
        // speed and throttle, so we'll map it from it's native [-1,1] to a simple scale
        // factor of [0,1] ...
        FlightStick joystick = Hardware.HumanInterfaceDevices.logitechAttack3D(1);
        ContinuousRange throttle = joystick.getThrottle();
        ContinuousRange sensitivity = throttle.map(t -> (t + 1.0) / 2.0);
        driveSpeed = joystick.getPitch().scale(sensitivity::read); // scaled
        turnSpeed = joystick.getRoll().scale(sensitivity::read).invert(); // scaled and i
        Switch trigger = joystick.getTrigger();
        // Get the RoboRIO's accelerometer ...
        ThreeAxisAccelerometer accel = Hardware.Accelerometers.builtIn();
        Accelerometer xAccel = accel.getXDirection();
        Accelerometer yAccel = accel.getYDirection();
        Accelerometer zAccel = accel.getZDirection();

        VoltageSensor battery = Hardware.powerPanel().getVoltageSensor();
        CurrentSensor current = Hardware.powerPanel().getCurrentSensor();

        Strongback.dataRecorder()
                  .register("Battery Volts",1000, battery)
                  .register("Current load", 1000, current)
                  .register("Left Motors",  left)
                  .register("Right Motors", right)
                  .register("Trigger",      trigger)
                  .register("Throttle",     1000, throttle::read)
                  .register("Drive Speed",  1000, driveSpeed::read)
                  .register("Turn Speed",   1000, turnSpeed::read)
                  .register("X-Accel",      1000, xAccel::getAcceleration)
                  .register("Y-Accel",      1000, yAccel::getAcceleration)
                  .register("Z-Accel",      ()->zAccel.getAcceleration()*1000));
        ...

    }
    ...
}
```

All of the sensors' `double` values are scaled by 1000 since they are converted to an integer when recorded. This truncates the values and reduces the overall of data recorded in the file, and can easily be converted back to the correct value during post-processing.

The "Z-Accel" channel is registered using a lambda rather than a method reference, showing how one can compute a value in a registered function.

Look at the "Trigger" channel, which records the state of the trigger at every cycle. Since the trigger's state changes quite infrequently (at least compared to the 50-200 times per second that the data recorder captures measurements), its much more efficient to use the event recorder discussed in the next section.

EventRecorder

Strongback's event recorder captures non-continous or infrequent events that your code supplies directly via the `org.strongback.EventRecorder` interface:

```
@ThreadSafe
public interface EventRecorder extends Executable {

    /**
     * Record an event with the given identifier and event information.
     *
     * @param eventType the type of event; may not be null
     * @param value the event details as a string value; may be null
     */
    public void record(String eventType, String value);

    /**
     * Record an event with the given identifier and event information.
     *
     * @param eventType the type of event; may not be null
     * @param value the event detail as an integer value; may be null
     */
    public void record(String eventType, int value);

    /**
     * Record an event with the given identifier and event information.
     *
     * @param eventType the type of event; may not be null
     * @param value the event detail as an integer value; may be null
     */
    default public void record(String eventType, boolean value) {...}

    /**
     * Return an {@link EventRecorder} implementation that does nothing.
     *
     * @return the no-operation event recorder; never null
     */
    public static EventRecorder noOp() {...}
}
```

The `eventType` parameter provides a logical name that is meaningful to you, and thus is very similar to the channel name used when registering functions with the `DataRecorder`. Unlike the `DataRecorder` that is always set up in the `robotInit()` method, the `EventRecorder` is used when the robot is operating and code wants to capture some "event".

We ended the previous section saying that capturing the state of a trigger is better done with the event recorder rather than the data recorder. Here's the same example we used in that section, except the trigger state change is captured as an event. We use the `Switch Reactor` to asynchronously detect the state change:

```
public class SimpleTankDriveRobot extends IterativeRobot {

    ....
    private TankDrive drive;
    private ContinuousRange driveSpeed;
    private ContinuousRange turnSpeed;

    @Override
    public void robotInit() {
        ...
        Motor left = Motor.compose(Hardware.Motors.talon(1), Hardware.Motors.talon(2));
        Motor right = Motor.compose(Hardware.Motors.talon(3), Hardware.Motors.talon(4)).i
        drive = new TankDrive(left, right);

        // Set up the human input controls for teleoperated mode. We want to use the
        // Logitech Attack 3D's throttle as a "sensitivity" input to scale the drive
        // speed and throttle, so we'll map it from it's native [-1,1] to a simple scale
        // factor of [0,1] ...
        FlightStick joystick = Hardware.HumanInterfaceDevices.logitechAttack3D(1);
        ContinuousRange throttle = joystick.getThrottle();
        ContinuousRange sensitivity = throttle.map(t -> (t + 1.0) / 2.0);
        driveSpeed = joystick.getPitch().scale(sensitivity::read); // scaled
        turnSpeed = joystick.getRoll().scale(sensitivity::read).invert(); // scaled and i

        // Capture the state change of the trigger ...
        Strongback.switchReactor().
                  .onTriggered(joystick.getTrigger(),
                               ()->Strongback.eventRecorder().record("Trigger",true))
                  .onUnTriggered(joystick.getTrigger(),
                                 ()->Strongback.eventRecorder().record("Trigger",false));

        // Get the RoboRIO's accelerometer ...
        ThreeAxisAccelerometer accel = Hardware.Accelerometers.builtIn();
        Accelerometer xAccel = accel.getXDirection();
        Accelerometer yAccel = accel.getYDirection();
        Accelerometer zAccel = accel.getZDirection();

        VoltageSensor battery = Hardware.powerPanel().getVoltageSensor();
        CurrentSensor current = Hardware.powerPanel().getCurrentSensor();

        Strongback.dataRecorder()
                  .register("Battery Volts",1000, battery)
                  .register("Current load", 1000, current)
                  .register("Left Motors",  left)
                  .register("Right Motors", right)
                  .register("Throttle",     1000, throttle::read)
                  .register("Drive Speed",  1000, driveSpeed::read)
                  .register("Turn Speed",   1000, turnSpeed::read)
                  .register("X-Accel",      1000, xAccel::getAcceleration)
                  .register("Y-Accel",      1000, yAccel::getAcceleration)
                  .register("Z-Accel",      ()->zAccel.getAcceleration()*1000));
        ...
    }
    ...
}
```

Of course, in the function we supplied to the `SwitchReactor` we could do more than just record the trigger state change. For example, we might also want start a command.

Recording commands

By default Strongback automatically records the state changes of each `Command` as it transitions through its lifecycle. This can be helpful to understand and explain changes in continuous data or other events. For example, if you have a command that runs the compressor, post-processing a log that includes commands can help explain an increase in the current load and decrease in the battery voltage.

Turning off recording

The data and event recorders do consume resources, so if you decide to not use this feature be sure to configure Strongback to disable the recorders. Again, this is done in the `robotInit()` method *before* you call `Strongback.initialize()` :

```
public class SimpleTankDriveRobot extends IterativeRobot {
    ....
    @Override
    public void robotInit() {
        // Set up Strongback using its configurator. This is entirely optional, but we wo
        // record data, events, or commands, so we'll turn them off. All other defaults a
        Strongback.configure()
                  .recordNoEvents()
                  .recordNoCommands()
                  .recordNoData()
                  .initialize();
        ...
    }
    ...
}
```

Executor

Strongback's executor framework simplifies writing *asynchronous* code that runs efficiently on the RoboRIO. For the most part, the executor is used by Strongback itself, and is not something most users will need. Regardless, if your robot code has a component that has to run reliably in the background, consider running it with Strongback's executor rather than creating a new thread.

Concepts

The executor framework involves two basic concepts. An *executable* component is anything that needs to be run periodically, and the *executor* is the thing that tracks, schedules, and runs executables at the appropriate times.

To create your own executable component, simply implement the `org.strongback.Executable` interface:

```
@FunctionalInterface
public interface Executable {
    /**
     * Perform an execution at a given moment within the robot match.
     *
     * @param timeInMillis the time in the match (in milliseconds) when this execution is
     */
    public void execute(long timeInMillis);
}
```

Unlike Java's standard `java.lang.Runnable` or `java.util.concurrent.Callable` interfaces that are designed to be run just one time, Strongback's `Executable` is designed for its `execute(…)` method to be called *multiple times*. The `execute(…)` method takes a single argument that tells the implementation the instant in time that the method is being called that some `Executable` implementations may find very useful. Of course, feel free to ignore the time if you don't need it.

In order to run your `Executable` implementation, you have to create an instance and *register* it with Strongback's global executor:

```
Executable myComponent = ...
Strongback.executor().register(myComponent);
```

Strongback will automatically stop your component from running when it is shutdown, but you can always explicitly *unregister* your component before then:

```
Strongback.executor().unregister(myComponent);
```

Why use it?

Strongback's single executor is designed to run as reliably and regularly as possible on the JVM. This is always challenging on the JVM where garbage collection can sometimes stop the world. But the relationship between Java threads and processor cores is intentionally very loosely defined, allowing the operating system and JVM to decide when to give each thread some processing time. Ignoring other things that the operating system might be doing, if we have *as many or fewer* threads than cores then all of those threads can potentially run all of the time. However, if our JVM runs *more* threads than cores, then we're guaranteed that some of the threads will be preempted at least some of the time.

Strongback attempts to minimize the number of threads to maximize the likelihood that all threads will always be running. We know the robot's "main" thread is used to periodically call the autonomous, teleoperated methods of the `IterativeRobot` class. So Strongback adds only *one* more thread that does work every *n* milliseconds with very little variation, and it uses a number of techniques to try to prevent preemption or thread context switches.

If we only allow ourselves 2 threads, and one of those is the "main" thread, then we have to do all of our asynchronous work on that single thread. This is why Strongback's command framework, data and event recorders, and switch reactor *all* use the executor framework.

Gotchas

There are a few things to be aware of when implementing and registering your own `Executable` instances.

First, by default Strongback's executor will run on a 20 millisecond (50 Hertz) rate, which means that *all* of the `Executable` instances must complete within 20 milliseconds or the executor will fall behind. Strongback's executor thread monitors the JVM time, so even though it might fall behind for one or a few cycles, as long as subsequent cycles are shorter than 20 milliseconds the thread will quickly catch up and get back on its schedule. Of course, you can configure the execution period to be something larger or smaller than 20 milliseconds, so be sure that you adjust it if you find that the executor falls behind on your robot and you see error messages in Strongback's log that look like this:

```
ERROR: Unable to execute all activities within 20 milliseconds!
```

Secondly, try to run as few things asynchronously as possible. If you’re not using the data recorder or event recorder features, be sure to configure Strongback to not record them. Only register those `Executable` implementations that you really need.

Logging

Although we recommend using a dedicated logging framework — even the JDK logging. However, sometimes that's overkill and you just want to output a handful of messages but want a bit more control than what `System.out` or `System.err` provide.

Strongback includes a very simple and lightweight application logging framework that you can use in your robot code. There are not many features, but it does allow you to easily generate error, warning, informational, debug, and trace messages. It also lets you easily control which level of log messages you want to capture, and which you want to discard. All captured messages are written to `System.out`. Like we said, it's a very simple framework.

Using the logger

You can use the Strongback logger in any of your classes:

```
Strongback.logger().warn("Something may have gone wrong!");
```

or

```
try {
    ...
} catch ( Throwable t ) {
    Strongback.logger().error(t,"Whoa, what happened?!");
}
```

You can also pass a *context* to the `logger(…)` method, too. Strongback's default logger doesn't use this, but if you hook up Strongback to use a different logging framework you might find it useful. Here's an example that obtains the logger for the `MyCommand.class` logging context:

```
Strongback.logger(MyCommand.class).debug("MyCommand executed once and completed");
```

Control the log level

Like many other logging frameworks, Strongback supports multiple levels of messages: ERROR, WARN, INFO, DEBUG, and TRACE. These are ordered, so enabling INFO level messages will include ERROR and WARN messages as well, but will discard DEBUG and TRACE.

By default Strongback captures INFO level and above, but you can change that when you configure Strongback in `robotInit()` :

```
public class SimpleTankDriveRobot extends IterativeRobot {
    ....
    @Override
    public void robotInit() {
        Strongback.configure()
                  .useSystemLogger(Logger.Level.DEBUG)
                  .initialize();
        ...
    }
    ...
}
```

Using a custom Logger

If you want to send your log messages somewhere other than `System.out` , you can always create your own implementation of the `org.strongback.Logger` interface, and register it when initializing Strongback:

```
public class SimpleTankDriveRobot extends IterativeRobot {
    ....
    @Override
    public void robotInit() {
        Logger myLogger = ...
        Strongback.configure()
                  .useCustomLogger(context->myLogger)
                  .initialize();
        ...
    }
    ...
}
```

Remember that you now have control over which log messages are captured, since all messages will be sent to your `Logger` implementation.

If you look carefully, the `useCustomLogger` method takes a function that returns a `Logger` given a context parameter. The above lambda always uses `myLogger` , but you can make it as complicated as you want.

Contributing

The Strongback project is operated as a community-centric open source project. Although founded by FRC Team 4931, we hope that many teams want to use Strongback and contribute to our community. Everyone is welcome in our community.

Conduct

Strongback is dedicated to providing a harassment-free community for everyone, regardless of gender, sexual orientation, disability, physical appearance, race, religion, or tabs vs spaces preference. We do not tolerate harassment of community members in any form. Harassment includes offensive verbal comments, sexual images, deliberate intimidation, stalking, sustained disruption, and inappropriate physical contact, and unwelcome sexual attention. Community members asked to stop any harassing behavior are expected to comply immediately.

Contribute

You can contribute to Strongback by using it, asking or answering questions, reporting problems, writing documentation, fixing bugs, discussing plans, and developing new features.

To participate in any capacity, please join our project mailing lists.

- User group - Join this if you want to ask a question about Strongback. You can read and search it without signing up.
- Developer group - Join this if you want to discuss development, features, functionality, design, or architecture.

You may also consider following @strongbacklib on Twitter.

Code

All Strongback code is hosted in our GitHub repositories. Have a look, get the latest code and build it locally, or Learn how to contribute code Strongback's coding conventions

Physical test platforms

We’d like to open source the designs for a series of hardware bench tests that will help us test Strongback. The idea is that we’d publish specifications and CAD drawings for each test platform, and then also open source the test code for each platform. Multiple FRC teams could volunteer to build one or several physical test platforms, download and install the appropriate test code on the platform, run the tests, and report the results and any problems or issues. Using this approach, the whole Strongback community can band together to significantly test and verify each release, and make Strongback that much better for FRC teams to use on their competition robots.

Website

Our website is under construction and will be available soon. Hopefully we can whip it into some decent initial state that is publishable and then easy for others to contribute to documentation or website design.

Share

Talk about the Strongback community, write a blog post, or tweet about it using the `#StrongbackLib` hashtag or mentioning @StrongbackLib in your tweets!

History

In the fall of 2014, FRC Team 4931 was getting ready for their second competition season. Their rookie year was a lot of work, but the programming team decided that for the upcoming season they wanted to do far more testing of their code. After looking at different approaches, they decided that they wanted an *abstraction layer* to hide as much detail as possible about the underlying hardware. Thus, the ideas behind Strongback were born, and the team worked very hard that season to make off-robot testing a reality.

At the end of the 2015 season, the lead programmer and one of the mentors presented their work at the 2015 FIRST World Championships Conference. During the conference, they offered to open source their framework so other teams could use it. Not only did many teams like the idea, quite a few offered to help. So later that summer, Team 4931 extracted the code from their 2014 codebase, made some improvements, and created the Strongback open source project.