Java Style Guide

# 1. Software Structure

This is a brief about the software code structure and organization that will be used this year for the core robot code.

## 1.1. Package Organization

All package names will be preceded by the following:

edu.ahs.spaceraiders.<name of robot>

Each sub-mechanism will have its own sub-package under this, as will the input classes. In the root package goes the main classes for the robot, including the starting class, Robot and likely a Port enum (as well as some other classes as seen fit).

## 1.2. Design Patterns

### MVC

We will not concern ourselves with the details of this pattern. Just know that we have three major, distinct types of classes in our software: controllers, output classes, and input classes. Data comes into the input classes, is sent via the listener pattern described below, and is processed by the controllers for each mechanism, which then use that information to control the actuators (motors, pistons, compressors, etc.) via the output classes.

All controller inputs are suffixed by Policy. So a climber controller class should be named ClimberPolicy.

Output classes are prefixed by Handle. Sometimes, a single mechanism may need more than one output class. The names correspond to the physical mechanism's name; for example, HandleShooter or HandleFinger.

There is one special input class: HumanInput.

The creation of controllers and input class instances is delegated to the main Robot class. Controllers create their own output classes.

### Listeners

In this story, there are two main characters, the listener and the input. The input has some important information; the listener is anyone who wants to know that information. But the input is very choosy with whom it talks to; it won't talk to just anyone off the street. So first, any listeners must satisfy some requirements, perhaps be able to wear clean clothes, bring it a gift, and walk its dog for it. Then, once the listener has fulfilled those requirements, the listener has to go and initiate a conversation with the input. Then, the listener is not allowed to speak, only listen. Whenever the input gets new information, it will tell it to the listener, the listener has to go do one of the things it was required to do and walk the input's dog.

In code, this story is a commonly used design pattern in object-oriented programming called listeners. There are two important classes: an input class (i.e. a sensor or human input device such as a button or joystick) and another class who wants to know the status of the input. The class who wants to know must implement an interface, which is analogous to signing a contract, which says that he must have a certain method(s) in his declaration. Then the class will register themselves with the input class, saying that they would like to listen to them. Whenever the input class receives new data, it will “fire an event” object to all registered listeners via the method(s) defined in the interface, and the class will handle the new data it has just received.

Below is a worked example of the concept using potentiometers (but w/o hysteresis, etc.):

public class PotentiometerEvent() {

private double potVal;

public PotentiometerEvent(double potValue) {

this.potVal = potValue;

}

public double getPotentiometerValue() {

return potVal;

}

}

public interface PotentiometerListener {

void onPotentiometerChange(PotentiometerEvent);

}

public class Potentiometer {

private List<PotentiometerListener> potListeners =

new ArrayList<>();

private AnalogPotentiometer pot =

new AnalogPotentiometer(Port.POTENTIOMETER);

public Potentiometer() { /\* initialization \*/ }

public void handleEvents() {

PotentiometerEvent e =

new PotentiometerEvent(pot.get());

for (PotentiometerListener l : potListeners)

l.onPotentiometerChange(e);

}

public void addListener(PotentiometerListener listener) {

potListeners.add(listener);

}

}

public class MechanismPolicy implements PotentiometerListener {

private HandleMechanism mech = new HandleMechanism();

private double potValue = 0d;

public void onPotentiometerChange(PotentiometerEvent e) {

potValue = e.getPotentiometerValue();

}

public class Robot extends IterativeRobot {

private Potentiometer inPot = new Potentiometer();

private MechanismPolicy polMech = new MechanismPolicy();

public void init() {

inPot.addListener(polMech);

}

public void telop() {

inPot.handleListeners();

// execution code

}

}

# 2. Key Conventions

## 2.1. Identifier Naming

All identifiers should have meaningful, concise, and self-documenting names.

### Packages

Packages are in all lowercase with no underscores anywhere.

### Classes

Classes are always named in UpperCamelCase and are nouns, reflecting the object orientation of the language.

### Methods & Variables

Methods and variables should be typed in lowerCamelCase. Methods are almost always going to be verbs; variables are usually nouns or adjectives.

### Constants

Constants are “variables” which are both static and final. Constants should be typed in CONSTANT\_CASE, with underscores between words. A set of related constants should be made into an enum class, which can be either privately in the class or in a separate file, depending on the scope of the constants.

### Use of Hungarian Notation

To those unfamiliar to the C/C++ world, Hungarian notation is a method of earmarking private instance variables so that it's easier to tell what object they are. To do this, simply add on a prefix to the beginning of the variable name. Here is a list of prefixes we commonly use to integrate.

|  |  |
| --- | --- |
| PREFIX | MEANING |
| in- | Input device |
| pol- | Mechanism policy/controller |

To see an example, here is a snippet of code from last year's Robot class:

...

DrivePolicy polDrive = new DrivePolicy();

ClawPolicy polClaw = new ClawPolicy();

ShooterPolicy polShooter = new ShooterPolicy();

PolicyCoordinator polPolicy = new PolicyCoordinator();

HumanInput inHID = new HumanInput();

ShooterInput inShooter = new ShooterInput();

ClawInput inClaw = new ClawInput();

...

Note that the input system organization shown here is not necessarily reflective of the current year's system.

Except for the use of these two prefixes in the main robot class, the use of Hungarian is completely optional, as long as variables are being given proper, meaningful names.

## 2.2. Public v. Private v. “Package-Private”

The general rule for scope is that entities should be in the most restricted scope that they can. If another class won't need access to that enum/variable/method, then keep those things to yourself.

### Use of Package-Private

To specify a class as package-private (only the classes within the same package can access it), simply do not put any identifiers there. Within a sub-mechanism's package, most classes will be package-private to the other mechanisms because they do not need access to the internal workings of that particular mechanism.

## 2.3. Common Structures

### Enum

A enum, in its most basic form, is a list of constants. An enum in Java is powerful in many ways; but for our purposes, an enum allows constants that don't have to be associated with a number, or value. They're just simply constants. An enum will be used for ports, and often in conjunction with state machines (see below).

public enum Port {

WINDING\_WINCH(1),

DRIVE\_MOTORS\_LEFT(2),

DRIVE\_MOTORS\_RIGHT(3);

int port = -1;

private Port(int portNum) {

this.port = portNum;

}

}

### While Loops

Forbidden, do not use. Usage is closely monitored and is not at all encouraged for a looping/cyclic program. Use for-loops instead.

### For Loops

In comparison with while loops, for loops are much more controlled and finite. With the introduction of Java SE 8 into FRC, there are at least 3 common methods of iterating over a list (we don't really use arrays).

METHOD ONE

for (int i = 0; i < list.size(); i++) {

list.get(i);

// do stuff with element

}

METHOD TWO (Iterators)

Item itm;

for (Iterator itr = list.iterator(); itr.hasNext(); itm =

itr.next();) {

// do stuff with item

}

METHOD THREE (Enhanced For Loop)

for (Item itm : list) {

// do stuff with item

}

Many of you will likely choose to use the last method listed here.

### State Machines

Though not technically a structure in the language of Java, state machines are commonplace and provide a powerful tool and medium to convey human logic into code. There are many types, which can be *roughly* categorized: cyclic and branching are the major forms that you will be likely to see, however state machines come in many shapes and designs.

A state machine is basically an algorithm (some logical piece of code) which keeps track of the state of the system and proceeds accordingly. There are two components to state machine implementation: command execution and a current state check, which sums up to a 4 step process:

1. check current state
2. execute according to the current statement
3. advance state (or not, depending on the setup of the system)
4. be able to repeat when called on (by some higher authority class)

A **cyclic state machine** (deterministic) is one that is composed of steps which loop back around at the end. These are the easier of the three to implement, but should still be designed carefully with the physical mechanism in mind.

A **decision tree** or a **branching state machine** (non-deterministic) is one that will require a set of decisions in order to complete the task. (Tormentum's claw policy was a complex branching state machine.)

## 2.4. Miscellaneous

* Space out your code into logical segments; this will help others to see your logic in the code.
* Almost all of the time, you should place classes in separate files.
* Use double for decimals and int for integers.
* Occasionally, it will be beneficial to use a smaller integer type. Ports, for example, are never negative nor above ~20.

# 3. Javadocs

Note: Some of these conventions apply only to the particular type of software we will be developing, not necessarily to all software in general.

Javadocs allow a programmer to document code in such a way that the documentation can be accessed from other classes in an IDE. One can also create a full-page javadoc for a class (but in robotics, that will often be overkill). **Javadocs only need to be written for non-self-documenting methods or classes.**

Javadocs look like comments, but have an extra asterisk (\*) in the beginning.

/\*

This is a normal comment. Boring!

\*/

/\*\*

\* This is a javadoc comment. Notice the extra asterisk in the

\* beginning. Here I'm using extra asterisks at the beginning of

\* each of these lines so that the text and the symbols look

\* more uniform. Most IDEs will do this for you automatically.

\*/

## 3.1. Classes

Often times, all that is needed for a class javadoc is the name of the author of the class and possibly a brief description if the purpose of the class is not obvious. There is *no* need for license info, etc. See an example of a class javadoc below:

/\*\*

\* This class handles the semantics of working with the

\* drive train. (Even this is a little unnecessary.)

\*

\* @author TimLin

\*/

## 3.2. Methods

All javadocs for methods have a brief description of what the method does or returns, but they can optionally have @param statements or @return statements, if they require parameters or return objects, respectively. Again, it is not necessary to write javadocs for self-documenting methods. See examples below:

// does not require a javadoc

public List<Object> getObjectList() {

return objlist;

}

/\*\*

\* Returns the current command from the pending command stack,

\* which are constants from the enum Command. If no commands,

\* returns Command.NO\_COMMAND.

\*

\* @return current pending command

public Command getCurrentCommand() {

Command cmd = (!cmdlist.isEmpty()) ?

cmdlist.getCommandFromIndex(cmdIndex) :

Command.NO\_COMMAND;

cmd.setPending(false);

return cmd;

}

/\*\*

\* Selects the next phase of the state machine to use.

\*/

public void updateState() {

switch (state) {

case 1:

shooter.startWinchMotor('f');

winchingTimer.reset();

winchingTimer.start();

break; // all-important break statement

case 2:

// TODO: finish implementation

break;

default:

break;

}

}