Aldric’s Car Java Client Design Document

# Specifications

(What does the program needs to do?)

**Brief Description:** The client will be used to control the car remotely over internet. It needs to send commands to the car, and receive the video feed from the car.

**Requirements:**

* Client can receive inputs from keyboard or a controller connected through a serial port;

# Design

(Design of the program)

## Broad Design

**Core Module** – Runs the main loop of the program; Connects to the inputs (SerialPort Controller or GuiKeyboard); Receives commands from the inputs; Sends commands to listeners (InternetConnection and Gui);

**Gui Module** – Grafical User Interface which shows the feed from the webcam, shows feedback from inserted commands, can adjust the settings and can build commands.

**Carpad Module** –Manages the Car Controller (Carpad);

**Command Module** – Manages the messages (commands) which are sent to the car, and how to access them;

**Commons** – Classes with general functionality, not exclusive to the project.

## Specific Design

### Core Module

**ClientCore:**

* Needs a CommandSource.
* Can have a reference to ClientGui.
* Has the main loop of the program.
* Can receive commands and be controlled by other objects, through ClientCoreProxy.

Methods:

*void run()*, Main loop of the program. While object is not terminated:

- It first checks if the CommandSource is connected. If it is not, sleeps for a while. If it is, reads a Command from CommandSource and processes the Command.

- Then, processes the RunnableQueue.

*ClientCoreProxy createProxy()*, Creates a new proxy to control this object.

*void setClientGui(ClientGui clientGui)*, Sets the ClientGui.

**ClientCoreProxy:**

* Interface which contains methods to interact with ClientCore. To be used when another threads (ex.: AWT Thread) wants to interact with ClientCore.

Methods:

**ClientCoreProxyImplementation:**

* Implements the Interface ClientCoreProxy.
* Can only be generated by ClientCore.
* Has a reference to the Runnable Queue of ClientCore.

**ClientCoreProxyDummy: (deprecated)**

* Dummy implementation which does nothing.

**CarpadSource:**

* Implementation of CommandSource which uses the Carpad through the serial port.
* Runs a CarpadReader in a separate thread, using an executor. After issuing the object to run, executor is shutdown. This way, it is possible to know if Carpad is disconnected by asking if executor is terminated.
* Initialization is done before CarpadReader runs.
* When connecting, tries to connect to the last known port it could connect to. Otherwise, uses a default port name.
* Uses CorePreferences to store and retrieve configuration data.

**CorePreferences: (done)**

* Implementation of EnumKey, with the keys and access to the Preferences of this program.

Enum Values: SerialPortName

Methods:

*static PreferencesEnum getPreferences()*, returns the PreferencesEnum object of this program.

**CoreListener:**

- Interface for objects listening to the ClientCore.

### Gui Module

**ClientGui:**

* Manages the GUI of the client.
* Needs a ClientCoreProxy to interact to.

Methods:

*static ClientCoreProxy getCoreProxy()*, returns a ClientCoreProxy associated with the ClientGui. It is static so that the object is accessible to all the Screens of the Client. The method only executes if it is executed from the AWT Thread.

// *static void setCoreProxy(),* sets the value of ClientCoreProxy. (DEPRECATED) Probably this is a bad idea. First construct ClientCore, then construct the ClientGui.

### Carpad Module

**CarpadReader: (done)**

* Accesses the data of Carpad hardware through serial port.
* Implements Runnable.
* Any method of the object shouldn’t be access while it is running in a thread, since they are not thread safe. It is the responsibility of the object using CarpadReader to know if it is currently running or not.

Methods:

ReadChannel<int[]> getReadChannel(), Returns a ReadChannel from where we can read the values from the Carpad. While the thread is inactive, the ReadChannel will be empty. If active and running in a thread, it will return the values from the Carpad. If for any reason the thread terminates, ReadChannel will become empty again.

*void run()*, The main loop of the object. On each cycle, reads the values from the serial port, builds an array with the read values and puts it in its WriteChannel. If the Channel is already full, the array is discarded. If connection is lost, or values could not be read, the loop terminates and resources are liberated..

[deprecated] *~~void deactivate()~~*, If in ACTIVE state, liberates resources associated with the object and puts the object in NOT\_ACTIVE state. As any of the methods of the object, it can only be called if object is currently not running in another thread.

*boolean* *activate(String portName)*, If in NOT\_ACTIVE state, attempts a connection to the given portName. If a connection is possible, object is put in ACTIVE state. Returns true if it could connect to the port. Before connecting, the method tests if the given portName gives a correct stream of inputs. After the object is ACTIVE, it can only go back to NOT\_ACTIVE if it is run in a thread and the thread terminates. The thread terminates if it is interrupted, or if there is an exception inside the thread (such as loss of connection).

**CarpadUtils: (done)**

**-** Utility methods for Carpad Module.

Methods:

*static String defaultSerialPortName()*, returns the default serial port name of the Carpad, according to operating system. Currently supports Windows and Linux (Debian).

*static String findCarpadPortName()*,Tries to find the name of the port to which the CarPad is connected. Retrieves a list of every serial port and tests them one by one. If a port tests positive, its name is returned. Otherwise, null is returned.

*static boolean testCarpadPort(String portName)*, Tests if the given port is connected to Carpad. It tries to connect to the port. If it connects, it looks for the kind of input that is expected from the Carpad. Returns true if it could connect to the port and could find the pattern of Carpad. False otherwise.

**CarpadSetup: (done)**

- Contains information about the Carpad, such as the value of the preamble, and by which order it sends the variables.

*static int PREAMBLE*, The value of the preamble.

*static CommandArrayVariable[] INPUTS*, The order by which the Carpad sends the inputs.

### Command Module

**CommandSource:**

* Interface to access commands from a command source;

Methods:

*boolean connect()*, If disconnected, attempts to connect the command source so it can send commands. Returns true if a connection could be made. When connected, a CommandSource sends Commands periodically.

*void disconnect()*, If connected, disconnects the command source so it stops sending commands. Frees all resources it requested when connected.

*boolean isConnected()*, returns true if the command source is currently sending commands. False otherwise.

*long commandPeriod()*, when a CommandSource is connected, it builds Command objects at a fixed rate, approximately. *commandPeriod()* returns this period, in milliseconds.

*Command readCommand()*, reads a Command object from the CommandSource. If there is no Command object available, the method blocks until there is a Command object available, or until it times out. If it times out, or CommandSource is not connected, returns a special Command with invalid status.

*void setTimeout(long timeout)*, sets the value of the timeout, in milliseconds, when reading a Command.

*long getTimeout()*, returns the value of the timeout, in milliseconds, when reading a Command.

*void setIntIdentifier()*, sets the IntIdentifier of this CommandSource.

*IntIdentifier getIntIdentifier()*, returns the IntIdentifier of this CommandSource.

**Command: (done)**

* Contains the information to send to the car.
* At this moment, assume 7 integer values: one counter and six angle values.
* Is immutable.

Methods:

*int[] getArray()*, returns a reference to the internal array of the command.

*int getValue(CommandVariable var)*, returns the respective value.

*boolean isValid()*, returns true if it is a valid Command, false otherwise. An invalid Command is used to signal that the CommandSource is not connected, nor producing valid Command objects.

*static Command buildInvalidCommand()*, returns a new Command object with invalid status.

**CommandArrayVariable: (done)**

* Enum representing the variables inside the command array. Currently, they are: COUNTER, ANALOG1, ANALOG2, ANALOG3, ANALOG4, WHEEL, and TRIGGER.

Methods:

*int getIndex()*, get the index associated with the variable.

*static int getArraySize()*, get the number of variables of a Command array.

**~~CommandStatus: (deprecated)~~**

(substituted by a boolean inside Command object)

* Enum representing the status of the Command object.

### Commons

**Identification/IntIdentifier: (done)**

* Generates integers, incrementally.

Methods:

*IntIdentifier(int startValue)*, Creates a IntIdentifier that will generate integers starting from the given value, inclusive.

*IntIdentifier()*,Creates a IntIdentifier that will generate integers starting from 0.

*int newInt(),* Returns a new int.

**RxtxUtils: (done)**

* Utility methods for RxTx Library (Serial and Parallel Port)

Methods:

*static SerialPort openSerialPort*(String portName, String appName), Tries to opens the Serial Port with name “portName”. If a connection was not possible, null is returned. “appName” is used to identify which application is connected to the port. If a connection is not possible, the event is logged.

*static List<String> getSerialPortList()*, Returns a list with the names of all the serial ports it could find in the system.

static boolean rxtxLibrariesExists(), Tests for the presence of RxTx dynamic libraries. Returns true if they are found, false otherwise. Currently, test only supports Windows. If another operating system is detected, returns true.

**Interfaces/EnumKey: (done)**

* Interface to enable the use of enums as keys, instead of String keys.

Methods:

*String getKey()*, returns a String which corresponds to the key for the current enum.

*String getDefaultValue()*, returns a String with the default value of the current enum.

**Preferences/PreferencesEnum: (done)**

* Wrapper for Preferences class, which uses EnumKey instead of Strings to access its values.

Methods:

*PreferencesEnum(Class<?> c, boolean local)*, Builds a PreferencesEnum. If local is true, fetches a UserNode for package of class c. If local is false, fetches a SystemNode for package of class c.

*String getPreference(EnumKey key)*, Returns the value associated with the specified key in this preference node. If there is no value associated with the specified key, the default value defined in EnumKey is returned.

*String getPreferenceReal(EnumKey key)*, Returns the value associated with the specified key in this preference node. If there is no value associated with the specified key, null is returned.

*void putPreference(EnumKey key, String value)*, Associates the specified value with the specified key in this preference node.

**Concurrent/WriteChannel: (done)**

* Wrapper for a bounded write-only Blocking Queue.

**Concurrent/ReadChannel: (done)**

* Wrapper for a bounded read-only Blocking Queue.