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

**Client Module** – Runs the main loop of the client; 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);

**Communication 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.

**CarpadCommandSource: (done)**

* 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. (check if this is working)
* Uses ClientPreferences to store and retrieve configuration data.

**ClientPreferences: (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.

**ClientProperties:**

* Implementation of PropertiesDefinition.
* Contains information about the Client’s Properties file.

**ClientListener:**

- 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 which variables it sends.

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

*static enum INPUT*, enum listing the inputs of the Carpad.

*static int NUM\_INPUTS*, the number of inputs of Carpad.

### Communication Module

**CommandSource: (done)**

* 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 connecting.

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

*long getCommandPeriod()*, 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 when the request is made, the method may block for a while, waiting until there is a Command object available, but it must never block indefinitely. If the method couldn’t read a Command object (e.g.: time out), null is returned. If CommandSource is not connected when this method is called, null is returned.

*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 setByteIdentifier()*, sets the *Byte*Identifier of this CommandSource.

*IntIdentifier getByteIdentifier()*, returns the *Byte*Identifier of this CommandSource.

**Command: (done) (interface)**

* Information to send to the car.
* Should be immutable.

Methods:

*int getValue(Command.Variable variable)*, returns the value mapped to the respective Command.Variable.

*enum Variable*, list of the variables of the Command object.

*int NUM\_INPUTS*, get the number of variables of Command.

(TODO): Probably it will need a enum FlagSetOneMask.

**~~CommandSetup: (done)~~ (interface) (deprecated) – merged with Command object**

* Contains information about a Command object. Enum representing the variables inside the command array. Currently, they are: COUNTER, ANALOG1, ANALOG2, ANALOG3, ANALOG4, WHEEL, and TRIGGER.

*static enum VARIABLE*, enum listing the variables of the Command object.

*static int NUM\_INPUTS*, get the number of variables of a Command array.

**CommandImplementation: (done)**

* Straightforward, simple implementation of Command.

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

(substituted by a boolean inside Command object)

* Enum representing the status of the Command object.

**CommunicationUtils:**

- Utility methods for communication.

Methods:

*byte[] toByteArray(Command command)*, transforms the contents of the Command into an array of bytes, for communication over internet.

*int toCommandValue(byte[] commandArray, Command.Variable variable)*, extracts information from a Command byte array.

### 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.

**PreferencesUtil:**

* Utility methods for Preferences-related classes.

Methods:

*static String generateProperties(PropertiesDefinition propertiesDef, Preferences preferences)*, Builds a String with represents a Properties file built from the information in PropertiesDefinition and the data in Preferences.

*static void loadPropertiesDefinition(PropertiesDefinition propertiesDef, Preferences preferences)*, loads the contents of a PropertiesDefinition into the Preferences.

**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.

**Interfaces/PropertiesDefinition:**

* A class that implements this abstract class can define what contents a Properties file should have, under the form of Section objects (comments and keys). This abstract class is used by PreferencesEnum to implement support for loading/saving properties files.

Methods:

*protected abstract void buildSections()*, builds the sections which will define the properties files. An implementation of this method should be composed by consecutive calls to addSection() methods.

*public abstract String getPropertiesFilename()*, returns the filename of the Properties file.

*public abstract EnumKey valueOf(String keyName)*, returns the enum constant of this type with the specified name, or null if the enum couldn't be found. The string must match exactly an identifier used to declare an enum constant in this type. (Extraneous whitespace characters are not permitted.)

*public boolean isAutoSaveEnabled()*, returns true if autosave is enabled. When autosave is enabled, any modifications in the PreferencesEnum are immediately reflected in the Properties file.

*public List<Section> getSections()*, returns a list with Section objects, which define the properties file.

*public void setAutoSave(boolean status)*, sets the status of autosave. By default, it is true.

*protected void addSection(String comment)*, helper method for adding a new section to a list.

*protected void addSection(String comment, EnumKey propertyName)*, helper method for adding a new section to a list.

Inner Classes:

**Section**

Methods:

*List<Section> getSections()*, returns a list with Section objects.

*String getPropertiesFilename()*, returns the filename of the Properties file.

*EnumKey valueOf(String keyName)*, Returns the enum constant of the EnumKey that is being backed-up with the specified name. The string must match exactly an identifier used to declare an enum constant in this type. (Extraneous whitespace characters are not permitted.)

**Interfaces/PropertiesDefinition/Section:**

* Class which stores a String[] and a EnumKey. The String[] represents comments, which should appear before the EnumKey.

**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.

*void addProperties(PropertiesDefinition properties)*, adds a PropertiesDefinition to the Preferences, so preferences are backed up by a properties file, along the mechanism for Preferences. After assigning a PropertiesDefinition, they influence Perferences in the following way:

* Right after being added, values from the properties file are loaded into the preferences; if properties file doesn’t exist, it is created with the current values of preferences.
* Changes in Preferences are reflected on the fields of the properties file.

*boolean saveProperties()*, if a PropertiesDefinition is associated with PreferencesEnum, updates the properties file with the current values. Returns true if the file could be successfully written.

*PropertiesDefinition getPropertiesDefinition(),* returns the PropertiesDefinition associated to this PreferencesEnum. If none is associated, null is returned.

*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.

// REMOVE

void loadProperties(Properties properties), loads and saves the contents of the Properties object in the Preferences.

**Concurrent/WriteChannel: (done)**

* Wrapper for a bounded write-only Blocking Queue.

**Concurrent/ReadChannel: (done)**

* Wrapper for a bounded read-only Blocking Queue.