Shared Library Design Document

# Specifications

(What does the program needs to do?)

**Brief Description:** Library which contains classes with functionality useful for more than one program. Usually, these classes are less general than what is provided by Java libraries. The objective is to provide narrower, practical code for common scenarios. For example:

* Logging methods to ease the use of Java Logger, to be used as a substitute of System.out and System.err.
* Methods to ease the use of Java Preferences and provide simple backup to Properties files of some parameters.
* Methods for basic manipulation of Files (reading, writing) without having to deal with exceptions.

# Design

(Design of the program)

## Broad Design

**Utility Classes –** Classes which only contain static methods, usually devoid of side effects. There are exceptions, such as LoggingUtils which makes changes that affect the whole system (ex.: redirecting the System.out to the Logger).

**Concurrent Package** – Objects for communication and operations between threads.

**Identification Package** – Objects for generation of unique keys, to identify other objects.

**Interfaces Package** – Interfaces used in the Shared Library.

**Logging Package** – Objects related to the use of the Java Logger API.

**Preferences Package** – Objects related to the use of the Java Preferences API.

**(Delayed until further development) Time Package** (GameBase) – Objects related to periodic or time-related execution.

**RxTx Package** – Separated project. Utility class with methods which use the RxTx Java Library. Includes a compiled jar from a third-party. Get the latest version here:

http://rxtx.qbang.org/wiki/index.php/Download

## Specific Design

### Utility Classes

* Reside on org.ancora.SharedLibrary.

**IoUtils: (AldricCarV2, JavaTools)**

Methods for quick and simple manipulation of files, folders and other input/output related operations.

***Methods:***

*static* *File* ***safeFolder****(String folderpath)*, given a string representing a filepath to a folder, returns a File object representing the folder. If the folder doesn’t exist, the method will try to create the folder. The method has the following steps: 1) if the folder exists, returns a File object representing its path; 2) if the folder doesn't exist, the method will try to create it, along with all the necessary folders; 3) if the folder could not be created, returns null; 4) if the path represented by folderpath exists but doesn't represent a folder, returns null. If a File object is returned, it is guaranteed that the folder exists.

*static* *File* ***safeFile****(String filepath)*, method to create a File object for a file which should exist. If the file given by filepath does not exist, or is not a file, returns null. The method has the following steps: 1) if the file exists, returns a File object; 2) if the path exists, but doesn't represent a file, null is returned;

*static File* ***create****(String filepath)*, method to create a File object for a file which should not exist. If the file given by filepath could not be created, returns null. The method has the following steps: 1) if the file exists, returns a File object; 2) if the file does not exist, an attempt is made to create an empty file. If successful, returns the File object. Otherwise, returns null.

*static String* ***read****(File file)*, given a File object, returns a String with the contents of the file. If the File object doesn't exist or doesn't represent a file, returns an empty string;

*static boolean* ***write****(File file, String contents)*, given a File object and a String, writes the contents of the String in the file, overwriting everything that was previously in the file, and returns true. If the File object doesn't exist or doesn't represent a file, nothing is written. If file could not be written, returns false.

*static boolean* ***append****(File file, String contents)*, given a File object and a String, writes the contents of the String at the end of the file, and returns true. If the File object doesn't exist or doesn't represent a file, nothing is written. If file could not be written, returns false.

*static Properties* ***loadProperties****(File file)*, given a File object, loads the contents of the file into a Java Properties object. If the File object doesn't exist, doesn't represent a file or could not be loaded into a Properties object, returns null.

*(consider) static List<String> readLines(File file),* given a File object, returns a List of Strings with the contents of the file, one String per found line. If the File object doesn't exist or doesn't represent a file, returns an empty List;

**LoggingUtils: (AldricCarV2, AldricCarV1, ASL)**

Methods for the Java Logger API.

***Variables:***

*final static PrintStream stdout*, stores a reference to the original System.out.

*final static PrintStream stderr*, stores a reference to the original System.err.

***Methods:***

*static void redirectSystemOut()*, redirects the System.out stream to the logger. Anything written to System.out is recorded as a log at info level.

*static void redirectSystemErr()*,redirects the System.err stream to the logger. Anything written to System.err is recorded as a log at warning level.

*static void setRootHandlers(Handler[] handlers)*, removes current handlers and adds the given Handlers to the root logger.

*static Handler buildConsoleHandler()*, builds a Console Handler which uses as formatter, ConsoleFormatter.

*static void setupConsoleOnly()*, automatically setups the root logger for output to the console.

**ParseUtils: (AldricCarV2, AldricCarV1)**

Utility methods for parsing of values which, instead of throwing an exception, return a default value if a parsing error occurs.

***Methods:***

*static int parseInt(String integer)*, tries to parse a String into a integer. If an exception happens, warns the user and returns a 0.

*static long parseLong(String longNumber)*, tries to parse a String into a long. If an exception happens, warns the user and returns a 0.

**PreferencesUtils: (AldricCarV2, AldricCarV1)**

Methods for the Java Preferences API.

***Methods:***

*static String generateProperties(PreferencesEnum preferences)*, if an object PropertiesDefinition is associated to the object PreferencesEnum, builds a String with represents the contents of a Properties file. Otherwise, warns the user and returns an empty String. The properties file is built from the data in the PreferencesEnum object and PropertiesDefinition object.

*static boolean loadPropertiesDefinition(PreferencesEnum preferences)*, if an object PropertiesDefinition is associated to the object PreferencesEnum, loads the contents of the corresponding Properties file into the Preferences.

*static boolean savePropertiesDefinition(PreferencesEnum preferences)*, if an object PropertiesDefinition is associated to the object PreferencesEnum, saves the content of the Preferences to a Properties file.

**TimeUtils: (ASL) (Maybe they should use the TimeUnit enumeration) (put just the method which transforms a number into an array with hours, minutes and seconds.)**

Utility methods related to time.

***Methods:***

*static int[] clockFormat(long elapsedTime, TimeUnit timeunit)*, returns an array with three elements, where based on the given amount of elapsed time, the first element represents hours, the second element represents minutes and the third element represents seconds.

**(Delayed until further development) BitUtils: (Microblaze, JavaTools)**

### Concurrent Package Classes (AldricCarV2)

Reside on org.ancora.SharedLibrary.Concurrent.

**ReadChannel:**

Sister object ofWriteChannel. Can only be created by a WriteChannel object, and represents the consumer end of a channel. Wrapper for a bounded read-only Blocking Queue.

***Methods:***

*T poll()*, retrieves and removes the head of this queue, or returns null if this queue is empty.

*T poll(long timeout, TimeUnit unit) throws InterruptedException*, retrieves and removes the head of this queue, waiting up to the specified wait time if necessary for an element to become available.

*T take() throws InterruptedException*, retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

**WriteChannel:**

Sister object ofReadChannel. Creates ReadChannel objects which are connected to the WriteChannel that created them. Represents the producer end of a channel. Wrapper for a bounded write-only Blocking Queue.

***Methods:***

*ReadChannel<T> getReadChannel()*, returns the corresponding ReadChannel of this WriteChannel object.

*boolean offer(T e)*, inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and false if no space is currently available. When using a capacity-restricted queue, this method is generally preferable to BlockingQueue.add, which can fail to insert an element only by throwing an exception.

*boolean offer(T e, long timeout, TimeUnit unit) throws InterruptedException*, inserts the specified element into this queue, waiting up to the specified wait time if necessary for space to become available.

*void clear()*, empties the WriteChannel.

### Identification Package Classes (AldricCarV2)

Reside on org.ancora.SharedLibrary.Identification

**ByteIdentifier:**

Generates bytes, incrementally, which can be used to uniquely identify objects.

***Methods:***

*byte newByte()*, returns a new byte.

**IntIdentifier:**

Generates integers, incrementally, which can be used to uniquely identify objects.

***Methods:***

*int newInt()*, returns a new integer.

### Interfaces Package Classes (AldricCarV2, ASL.GameBase)

Reside on org.ancora.SharedLibrary.Interfaces

**EnumKey:**

Enables 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.

### Logging Package Classes

Reside on org.ancora.SharedLibrary.Logging

**LoggingOutputStream:**

An OutputStream that writes contents to a Logger upon each call to flush().This class is used by LoggingUtils methods to redirect the System.out and System.err streams.

***Methods:***

*void flush() throws IOException*, upon flush() write the existing contents of the OutputStream to the logger as a log record.

**ConsoleFormatter: (formatter used for presenting logging information on a screen)**

Extension of Formatter class, used for presenting logging information on a screen.

### Preferences Package Classes

Reside on org.ancora.SharedLibrary.Preferences

**PreferencesEnum:**

Wrapper for Preferences class, which uses EnumKey instead of Strings to access its values, and provides additional properties such as saving to a user-defined Properties file, through a PropertiesDefinition object.

***Methods:***

*void addProperties(PropertiesDefinition propertiesDefinition)*, adds a PropertiesDefinition to the object, so Preferences are backed up by a Properties file. After assigning a PropertiesDefinition, they influence Preferences: 1) when adding propertiesDefinition, if the Properties file indicated by the PropertiesDefinion object exists, the values from the Properties file are loaded into the Preferences; otherwise, a new Properties file is created with the current values of Preferences. 2) if *autosave* option is enabled in the PropertiesDefinition, everytime there is a change in a Preferences field, the change is immediately reflected on the corresponding field of the properties file.

*PropertiesDefinition getPropertiesDefinition()*, returns the PropertiesDefinition object 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. If there is a PropertiesDefinition associated with this PreferenceEnum, and the PropertiesDefinition has the *autosave* option enabled, everytime this method is called, the changes are reflected in the Properties file.

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

**PropertiesDefinition:**

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

***Methods:***

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

*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() protected methods.

*abstract EnumKey valueOf(String keyName)*, returns the EnumKey with the specified name. The string must match exactly an identifier used to declare the EnumKey constant in this type. (Extraneous whitespace characters are not permitted.)

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

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

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

### RxTx Package Classes (Optional)

Reside on org.ancora.SharedLibrary.

**RxtxUtils:**

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

***Methods:***

*static List<String> getSerialPortList()*, returns a list of Strings with the names of all the serial ports that could be found in the system.

*SerialPort openSerialPort(String portName, String appName)*, attempts to open the Serial Port with name “portName”. If a connection is 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 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 and shows warning.