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.

(está a mais, pode-se criar File com safeFolder e depois usar file.getPath) *static* *File safeFolder(File folder, String foldername)*, given a File representing a folder, and a String representing the name of another folder inside the previous folder, returns a File object representing the complete folder. The method has the following steps: 1) if the folder inside the first folder exists, a File with its path is returned; 2) if the folder doesn't exist, an attempt is made to create the folder and all the needed folders; 3) if unsucessful, returns null; 4) if the complete folderpath exists, but doesn't represent a folder, null is returned;

*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;

(está a mais, pode-se criar File com safeFolder e depois usar file.getPath) *static File safeFile(File folder, String filename)*, 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, null is returned. The method has the following steps: 1) if the file inside the folder exists, returns a File object; 2) if the folder doesn't exist, an attempt is made to create the folder and all the needed folders. If unsuccessful, returns null; 3) if the complete filepath exists, but doesn't represent a file, returns null;

*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.

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

**ParseUtils: (AldricCarV2, AldricCarV1)**

**PreferencesUtils: (AldricCarV2, AldricCarV1)**

**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.)**

**BitUtils: (Microblaze, JavaTools)**

### Concurrent Package Classes (AldricCarV2)

Reside on org.ancora.SharedLibrary.Concurrent.

**ReadChannel:**

**WriteChannel:**

### Identification Package Classes (AldricCarV2)

Reside on org.ancora.SharedLibrary.Identification

**ByteIdentifier:**

**IntIdentifier:**

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

Reside on org.ancora.SharedLibrary.Interfaces

**EnumKey:**

### Logging Package Classes

Reside on org.ancora.SharedLibrary.Logging

**LoggingOutputStream:**

### Preferences Package Classes

Reside on org.ancora.SharedLibrary.Preferences

**PreferencesEnum:**

**PropertiesDefinition:**

### 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.

### 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.