# Introduction

This document describes the design for Archiver, a Desktop software tool that remotely archives files on a local computer on a scheduled basis. This tool is designed to scale to meet the needs of advanced users to intermediate computer users. Archiver is desgined to run on MacOS X and Windows.

# Assumptions

* Higher level tools such as cron cannot be effectively integrated
* Consistent user interface across platforms to minimize development time, training and documentation.
* We want to leverage the in house Java development skills.
* Human readable file formats for advanced users.

# Document Scope

This document covers the basic design and testing of Archiver. It does not cover specific platform issues such as StartupItem/Service integration and installers.

# High Level Overview

At a high level, Archiver is composed of three main components:

1. Backend
2. UI
3. Shared Library

The Backend is responsible for deserializing the job list, scheduling, logging and executing the jobs and updating its current state based on the UI or OS events (such as shutdown, restart, etc…). The Backend would typically start up in its own process when a machine boots up, as a StartupItem on MacOS X and a Service under Windows. Backend uses human readable file formats to configure job settings so that advanced users can configure the system with a text editor.

The UI runs in a process separate from the Backend so that it can be started and restarted independently of the Backend process. It is not necessary to run the UI in order to use Archiver, but is provided to assist less advanced users. The UI communicates with Archiver over an RMI interface

The Shared Library component is a JAR that contains functionality that is common to the Backend and UI such as representations for the backup schedule. This component is shared between the Backend and UI to help ensure version compatibility.

# Detailed Design

## Backend Classes

### com.jcho.archiver.backend.Archiver

This class is responsible for initialization, which includes reading the **Jobs.yml** file that contains the scheduling information and the **JobStatus.yml** file that contains information about the last execution of the jobs and invoking the scheduler to execute the jobs. It sets up the log file that can be used for general debugging purposes. It must set up and maintain the RMI that allows communication to the OS and UI. The most important methods include:

* JobList getJobList() // returns a copy of the current set of Jobs
* setJobList(JobList) // sets and saves the current list of Jobs to execute
* JobStatus getJobStatus(String jobName) // gets the status for the given job
* shutdown() // shutsdown the service
* restart() // restarts the service
* String readLogEntry(long n) // reads the nth log entry
* long getLogEntryLength() // returns the number of entries in the log

Querying the RMI interface, reading the logs, JobList.yml, can test this class and JobStatus.yml files. com.jcho.jobs.Scheduler will have to be mocked so that events are triggered “on-demand” rather than waiting for the actual scheduled time. The mockup can also be constructed to ensure that shutdown and restart are correctly invoked.

The logger and scheduler facilities will be constructed to ensure Thread Safety.

### com.jcho.archiver.model.BackupJob

This subclass of Job represents a folder to be backed up and its corresponding destination. This class will utilize com.jcho.ZipUtil and org.apache.commons library to archive and ftp the data respectively. Its important members include:

* File sourceDirectory // source directory to archive
* org.apache.commons.httpclient.URI destinationURI // destination to store sourceDirectory

This class can be tested by invoking its run() method and determine whether or not the desired command was run.

### com.jcho.jobs.Job

This is an abstract class that implements Runnable and represents a job that has to be done. Its important members include

* String name; // name of the Job
* long startTime; // start time in Java milliseconds
* long endTime; // end time in Java milliseconds
* long repeatInterval; // repeat interval in Java milliseconds or 0 if no repeating is desired

Testing this class is primarily a matter of determining that the accessors access the member variables correctly.

### com.jcho.jobs.JobStatus

This is the status of a run Job. Its important members are:

* String name; // name of the Job
* Long lastRunTime; // last time the job was run in milliseconds
* String error; // null or an error message

Testing this class is primarily a matter of determining that the accessors access the member variables correctly.

### com.jcho.jobs.JobList

This is a list of jobs that is somewhat limits the access to the actual List<Job>. Its important members are:

* setJobs(List<Job> jobs); // sets the list of jobs to jobs
* List<Job> getJobs(); // returns a copy of the list of jobs
* addJob(Job job); // adds a job, replacing any job with the same name
* removeJob(String name); // removes the job with the given name

To test this class one must verify the effect of each method on the whole object.

### com.jcho.jobs.JobStatusList

This is a list of jobs that is somewhat limits the access to the actual List<JobStatus>. Its important members are:

* List<JobStatus> getJobStatuses(); // returns a copy of the list of jobs
* setJobStatuses(List<JobStatus>); // sets the current list of JobStatues
* addJob(Job job); // adds a job, replacing any job with the same name
* removeJob(String name); // removes the job with the given name

To test this class one must verify the effect of each method on the whole object. Thread safety will be ensured by synchronizing all public functions and using reflection to ensure this.

### com.jcho.jobs.JobListSerializer

This class is responsible for serializing the status of Jobs to the **Jobs.yml** file. It uses yamlbeans to perform the actual serialization. Its important static members include:

* serialize(JobsList, File outputFile)
* JobsList deserialize(File inputFile)

Serializing different JobStatusLists and then using a “generic” YML deserializer and comparing the corresponding hashes and sequences can test serialization. Deserializing different JobStatus.yml and comparing the results against the expected JobStatus object can be used to test deserialization.

Serialiaing and deserializing and comparing the results can test integration.

### com.jcho.jobs.JobStatusListSerializer

This class is responsible for serializing the status of Jobs to the **JobStatus.yml** file. It uses yamlbeans to perform the actual serialization. Its important static members include:

* serialize(JobStatusList, File outputFile)
* JobStatusList deserialize(File inputFile)

Serializing different JobStatusLists and then using a “generic” YML deserializer and comparing the corresponding hashes and sequences can test serialization. Deserializing different JobStatus.yml and comparing the results against the expected JobStatus object can be used to test deserialization.

Serialiaing and deserializing and comparing the results can test integration.

### com.jcho.jobs.Scheduler

The scheduler is responsible for executing Jobs periodically. It ensures that its setJobs() and getJobs() methods are Thread Safe and do not interfere with currently executing jobs. It has a JobList that it schedules using java.util.concurrent.ScheduledExecutorService. After a Job is executed or is terminated, it invokes a callback to notify other classes.