Archiver Design

Version 1

Jamie Cho

Table of Contents

2. Introduction 2

3. Assumptions 2

4. Document Scope 3

5. High Level Overview 3

6. Detailed Design 5

a. Backend 5

I. com.jcho.archiver.backend.Archiver 5

b. UI 6

c. Jobs Library 6

II. com.jcho.jobs.SchedulerSystem 6

d. com.jcho.jobs.SchedulerSystemService 7

III. com.jcho.jobs.Scheduler 8

IV. com.jcho.jobs.Job 8

V. com.jcho.jobs.JobStatus 9

VI. com.jcho.jobs.JobList 9

VII. com.jcho.jobs.JobStatusList 9

VIII. com.jcho.jobs.JobListSerializer 9

IX. com.jcho.jobs.JobStatusListSerializer 10

e. Backup Library 10

X. com.jcho.archiver.model.ArchiveJob 10

f. Utility Library 11

XI. com.jcho.util.logging.Logger 11

g. com.jcho.util.zip.ZipUtil 11

7. File Formats 11

h. Jobs.yml 11

i. JobStatus.yml 12

# 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.
* It is likely that Archiver’s functionality will be useful in other products including the actual backup facility and the ability to run generic jobs such as periodically running a shell script.
* Archiver only supports the backup but not the restore. The restore functionality can be added in the future.

# 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. This document does not address the future needs of beginners and the desirability to direct their business to partner services.

# High Level Overview

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

1. Backend
2. UI
3. Shared Archiver Library
4. Shared Jobs Library
5. Shared Utility 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 Archiver Library component is a JAR that contains functionality that is common to the Backend and UI. The most important functionality that it currently contains is the representation for an Archiving Job that includes the source and destination locations.

The Shared Jobs Library component is a JAR that contains functionality that is common to scheduling Jobs that are run on a periodic basis.

The Shared Utility Library is a JAR that contains functionality that is useful across multiple applications including ZipTool and Loggers.

# Detailed Design

## Backend

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

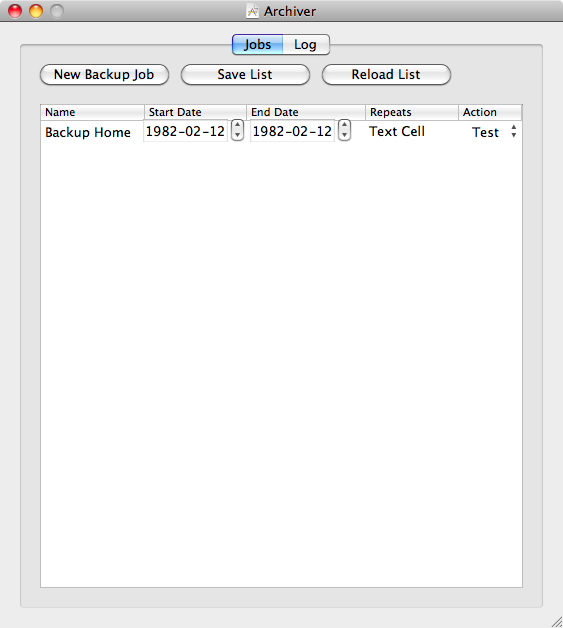
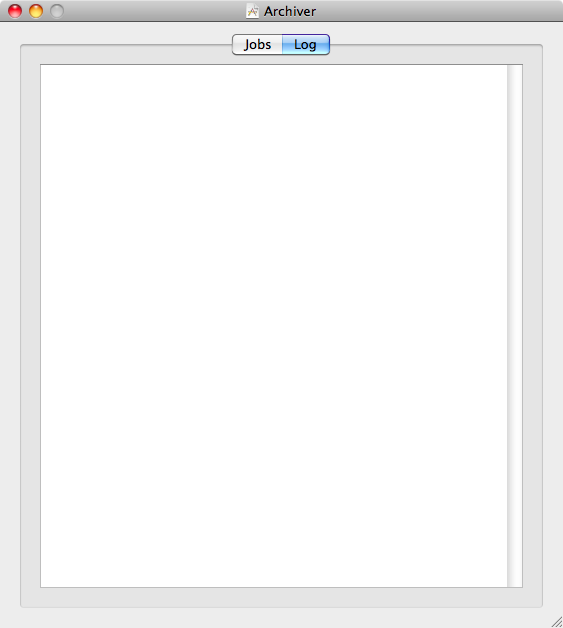
This subclass of com.jcho.jobs.SchedulerSystem provides a static main method that serves as the main entry point to the backend. It performs any additional customizations required for Archiver including starting the SchedulerSystemService.

Testing this class involves ensuring the correct files are created and read, ensuring that the RMI service is properly established. Mocking the Scheduler component and inserting Jobs that stress different behaviors can assist with integration testing.

Integration testing should ensure that the unzipped contents match the original contents. This should be done against the common unzip utilities on MacOS X and Windows to ensure the archive is readable across platforms.

Integration testing must also ensure that the actual ftpd contents are correctly uploaded in binary mode. This can be done through mocks or integrating Apache’s FtpServer.

## UI

The UI classes will exist in the **com.jcho.archiver.ui** package and implements a SWING based UI. The details of the UI will are not presented here, but it is conceptually simple to build and will consist of two main screens:

1. Jobs which allows the user to add, remove, edit, test and save jobs
2. Log which gives the user a view of the current Backend logs

When the UI launches, it will establish an RMI connection with the backend server and get a copy of the list of files and log file. If the connection cannot be established, it must inform the user and exit.

It will periodically invoke the RMI to update the log file. It also connects to the backend when it is refreshing and saving the Job list. When the user tests a given operation, the frontend directly invokes the Job.

The UI can be tested via QF-Test (see <http://www.qfs.de>) and by mocking the connection with the backend. Mocking ArchiverJob can effectively exercise the Test functionality.

## Jobs Library

The Backend and UI both share the Backup Library. This library could be used by other products that need generic scheduling capabilities.

### com.jcho.jobs.SchedulerSystem

This class implements a generic scheduler facility that loads and saves jobs to a jobs file and executes those jobs. This class is responsible for reading and writing the job files and using or initializing a logger. The default constructor will use Java properties for default configuration for logs and file locations. When reading the files, it is responsible for logging any failures to the Logger.

This class delegates the actual work to various components including JobListSerializer and JobStatusListSerializer, JobList, JobListState, Scheduler and Logger.

Testing this class involves ensuring the correct files are created and read. Mocking the Scheduler component and inserting Jobs that stress different behaviors can assist with integration testing.

## com.jcho.jobs.SchedulerSystemService

This class implements an RMI service that allows a client including the UI to communicate with a SchedulerSystem. It also provides basic O/S facilities to restart and shutdown that be typically be invoked by LaunchServices (MacOS X) or ServiceManager (Windows).

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[] readLogEntries(long n) // reads all log entries starting at the nth log entry index (0 based)

### 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 started, completed or terminated, it invokes a callback to notify other objects. It logs to the default Logger when a backup Job has started and ended and also logs additional information including failures. This class is responsible for managing the running Jobs and cancelling Jobs that have expired.

This is the most challenging class to test. Effectively testing it requires mocking ScheduledExecutorService and ensuring that it receives the expected inputs including dates, times, and runnables. Mocking a Logger or using com.jcho.util.logging.Logger can achieve testing the logging behavior. Mocking Jobs that exhibit different scenarios such as throwing, taking a long time or a short time can be passed into Scheduler. Final integration with ScheduledExecutorService can be tested by mocking jobs that are executed with short durations and repeat intervals.

### com.jcho.jobs.Job

This is an abstract class implements Runnable and Serializable 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. Testing must also sure that de/serialization also works with the JobListSerializer as well as testing the Serializable interface.

### 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. Testing must also sure that de/serialization also works with the JobListSerializer as well as testing the Serializable interface.

### com.jcho.jobs.JobList

This is a list of jobs implements Serializable 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. The Serializable interface must be tested by serializing and deserializng the 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. Must fail when it encounters an unknown version number. Serialiaing and deserializing and comparing the results can test integration.

The version number it uses for serialization must be incremented

### 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. Must fail when it encounters an unknown version number.

Serializing and deserializing and comparing the results can test integration.

## Backup Library

The Backend and UI both share the Backup Library. This library could be used by other systems that need scheduled backups.

### com.jcho.archiver.model.ArchiveJob

This subclass of Job that also implements Serializable 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. Substitutes %t for current time and escapes %% with %
* org.apache.commons.httpclient.URI destinationURI // destination to store sourceDirectory archive

This class can be tested by invoking its run() method and determine whether or not the desired command was run. Testing must also sure that de/serialization also works with JobListSerializer. The Serializable interface must also be tested by serializing and deserialiing. Mocking of the FTP facilities can be used to ensure that they are initialized as expected. Use against an actual FTP server can facilitate integration testing.

## Utility Library

This library implements facilities that may be useful in a wide variety of applications.

### com.jcho.util.logging.Logger

Logger is a subclass of java.util.logging.Logger that captures all output and makes it possible to both read and write to the log. When the log is initially created, it reads in all of the existing log entries. The important new methods are:

* String[] read(long n); //Reads all log entries starting at the nth index. Note that the number of indices is always growing after the log is constructed even after truncation. Any entry at or after n that has already been truncated is omitted form the output (that is, there is no Array entry made for it).

Testing involves reading and writing to the log, reconstructing the log. Inspection should ensure that readLog() must be ensured to be ThreadSafe but not block logging inordinately long for larger log files

## com.jcho.util.zip.ZipUtil

This class should be extended to better handle attributes, permissions and forks supported by HFS+ and NTFS as well as locked files. This functionality is not natively supported by Java and will require licensing a third party library. Implementing this support in house in C/C++ or launching preinstalled utilities for archiving. This requirement and possible solutions need further discussion.

This library requires testing. Archiver only makes use of the decompression

# File Formats

To facilitate expressivity AND ease of reading, Archiver and the SchedulerSystem uses YML to store the list of jobs and their statuses. Other common choices include XML and JSON were rejected because they are harder to read or lack comments.

## Jobs.yml

version: 1 # integer version number

jobs:

- !com.jcho.archiver.model.ArchiveJob

name: Backup Home

startDate: 2005/10/31T17:11:09 # ISO-8601

endDate: 2010/10/331T17:11:09 # ISO-8601

runs: daily #[once, daily, weekly, monthly, yearly]

src: /Users/jcho

dest: ftp://jcho:password@ftp.aws.com/home%t.zip

- !com.jcho.backup.job

name: Backup Applications

startDate: 2005/10/31T17:11:09 # ISO-8601

endDate: 2010/10/331T17:11:09 # ISO-8601

runs: monthly #[once, daily, weekly, monthly, yearly]

src: /Users/jcho

dest: ftp://jcho:password@ftp.aws.com/apps%t.zip

## JobStatus.yml

version: 1 #integer version number

jobStatuses:

- !com.jcho.jobs.JobStatus

name: Backup Home

lastRun: 2010/10/331T17:11:09 # ISO-8601 or null string

status: Failed. Incorrect password # could be null str

- !com.jcho.jobs.JobStatus

name: Backup Application

lastRun: 2010/10/331T17:11:09 # ISO-8601 or null string

status: "" # could be null str