Job Processor 3.0 Component Specification

1. Design

The Job Scheduling Component enables the timed execution of specified tasks. This functionality is similar to the unix cron utility and variants that exist on most operating system.   Users can schedule both one-time and repeating tasks.

In the second version of this component, the concepts of Job dependency and grouping were introduced to create a more robust capability. Jobs could be created that are dependent on other jobs, and to more easily perform jobs by grouping them. The third version, as proposed by this design, is to separate the scheduling part of the design from the processing part. This new scheduler will be solely responsible for managing the jobs, including their persistence, and will be moved to a separate component. The processor will then use that persistence to look up schedules.

The processor will take over the starting and stopping aspects of the Scheduler.

1.1 Design Patterns

The Scheduler implementations are used as **Strategies.** The TriggerEventHandler as it is strategically registered as an event handler with the job.

1.2 Industry Standards

None.

1.3

Required Algorithms

These algorithms are mostly untouched from 2.0. Where there are changes, these are indicated with light blue. The developer will thus be able to reuse most of the code in the 2.0 version.

1.3.1 *Scheduler Algorithm*

This algorithm is from 2.0. Modifications are in blue.

Purpose:
To schedule any job which has a specific schedule time.

Implementation:
The scheduler algorithm is implemented in the JobProcessor.startExecuteHeartBeat method, and is started by the JobProcessor.start method. There are two lists in the processor, the jobs list and the executing jobs list.
It uses java.util.Timer and TimerTask to setup the heartbeat thread.

```
executeTimer = new Timer();
        executeTimer.schedule(new TimerTask() {
                public void run() {
                    //sifts through jobs list, for expired jobs, removes them, for due
jobs, executes them
                    synchronized (jobs) {
                        GregorianCalendar rightNow = new GregorianCalendar();

                        for (int i = jobs.size() - 1; i >= 0; i--) {
                            Job job = (Job) jobs.get(i);
                            GregorianCalendar stopDate = job.getStopDate();

                            int executionCount = job.getExecutionCount();

                            if (((stopDate != null)
                                    && (rightNow.after(stopDate) ||
job.getNextRun().after(stopDate)))
                                  || (executionCount == 0)) {
                                // the job is expired, or reached its occurrence limit,
remove it.
                                jobs.remove(i);
                            } else if (!rightNow.before(job.getNextRun())) {
                                //the job is ready to start, records last run time
                                job.setLastRun(rightNow);

                                //calculates the next execution for the job

job.setNextRun(NextRunCalculator.nextRun(job.getStartDate(), rightNow,
                                        job.getIntervalUnit(),
job.getIntervalValue()));

                                if (executionCount != -1) {
                                    //if job is not a forever cycled job, decreases
executionCount

                                    job.setExecutionCount(executionCount - 1);
                                }

                                //starts the job and adds it into execution jobs list
                                Job executingJob = new Job(job);

                                    executingJob.start();
                                    executingJobs.add(executingJob);
                            }
                        }
                    } // end of synchronized (jobs)
                    //sifts through executingJobs list, for finished jobs, removes them,
and fires event if they
                    //are v2.0 jobs.
                    synchronized (executingJobs) {
                        for (int i = executingJobs.size() - 1; i >= 0; i--) {
                            Job job = (Job) executingJobs.get(i);

                            String status = job.getRunningStatus();

                            //if job is done
                            if (status.equals(ScheduledEnable.SUCCESSFUL) ||
status.equals(ScheduledEnable.FAILED)) {
                                //for ver2.0 job, fire the event on it
                                if ((job.getJobType() == JobType.JOB_TYPE_EXTERNAL) ||
job.isScheduledEnable()) {
                                    job.fireEvent(status);
                                }

                                //removes the completed job
                                executingJobs.remove(job);
                            }
                        }
                    }

                }
            }, 0, TIMER_INTERVAL);
```

1.3.2 *Reloader Algorithm*

This algorithm is new to 3.0.

Purpose:
To reload jobs at a regular time, to be set at construction time.

Implementation:
The reloading algorithm is implemented in the JobProcessor.startReloadHeartBeat method, and is started by the JobProcessor.start method. It uses java.util.Timer and TimerTask to setup this heartbeat thread.

```
 reloadTimer = new Timer();

//schedule a new task to reload the Jobs data
reloadTimer.schedule(new TimerTask() {
        public void run() {
            //gets all jobs
            Job[] retrievedJobs = null;
            retrievedJobs = scheduler.getJobList();

            if (retrievedJobs != null) {
                synchronized (jobs) {
                    //sifts through these jobs to get the ones that will be added.
                    List tempList = new ArrayList();

                    for (int i = 0; i < retrievedJobs.length; i++) {
                        Job retrievedJob = retrievedJobs[i];

                        //dependent job has no start date, should be filtered
                        if (retrievedJob.getStartDate() == null){
                            continue;
                        }

                        if (!retrievedJob.getActive()) {
                            //removes inactive jobs
                            jobs.remove(retrievedJob);
                            } else if (!jobs.contains(retrievedJob)) {
                                //adds if job is new
                                tempList.add(prepareJob(retrievedJob));
                            } else {
                                Job oldJob = (Job) jobs.get(jobs.indexOf(retrievedJob));

                                //else if the job already exists in the job list and it's modified
since last loaded
                                if
(!oldJob.getModificationDate().equals(retrievedJob.getModificationDate())) {
                                    tempList.add(prepareJob(retrievedJob));
                                    jobs.remove(oldJob);
                                }
                            }
                        }
                        //adds all updated jobs
                        jobs.addAll(tempList);
                    } // end of synchronized
                }
            }
        }, 0, reloadInterval);
```

Note

A cycle running Job may trigger its depdendent job to run multi times, thus it will cause serval Jobs will the same name exist in the Jobs list, however,

this will not make any trouble to the jobs' executions. While a job is inactive, it should be removed from the jobs list, and it will make sense that the dedpendent jobs of it will be removed as well, but it's not the requirement of current design.

1.3.3 Next job run time calculation at startup.

On startup the component needs to calculate the date and time that a job next needs to run. The date and time calculated should always be in the future, and should always be recalculated from the actual scheduling data (as opposed to the run-log data). The component should never attempt to make up execution times missed while not running or idle.

Examples:

Job A is scheduled to run every hour, starting at 0100 on 01 March. The component is started on 03 March at 0305. The next scheduled run time for Job A, after the component starts up, should be 0400 on 03 March. Job A then runs for a day; the scheduler is then shut down on 04 March at 0405, right after it’s 0400 run, and is not restarted until 04 March at 0730, missing time slots 0500, 0600, and 0700. The next scheduled run time should be 0800, as missed time slots are not made up.

Job B is initially scheduled as such: starting on 01 March at 0200, run every 30 minutes. The component is started and Job B runs several times. The last log entry is shown as 03 March, 0330, meaning that is when the job was last launched. The component is shut down, the job parameters are altered in the configuration file offline as such: time interval and unit is changed from 30 minutes to 1 hour (nothing else is changed), and the component is restarted at 0345. The component should recalculate the job based on the configuration file information, starting on 01 March and advancing until a date in the future is found using the new time increment of one hour, which would be 03 March at 0400. The log showing the last run time is ignored for these calculations (had it not, the next scheduled run time would have been 03 March at 0430).

Internally, GregorianCalendar is used to store and manipulate dates.
Note that if the job stop date/time falls in the past, it is not necessary to load the job in the processor.

The basic algorithm is:

```
For each job in configuration file.
  NextRun ← job start date/time.
    While NextRun <= currentTime and (NextRun <JobStopDateTime or JobStopDateTime is null)
            NextRun ← NextRun + jobRunInterval
```

Note that a null JobStopDateTime indicates there is no stop date for the job and it should therefore be re-scheduled indefinitely.

1.3.4

Trigger the job which has a dependency job to start

This algorithm is from 2.0. Modifications are in blue.

Purpose:

A Job in version 2.0 can be dependent on another job regarding the execution time. This algorithm shows how to trigger the dependent job to start.

Implementation:

This function is implemented using the TriggerEventHandler. When a job is retrieved from the scheduler, the processor will add a TriggereEventHandler to the job, so that, when this job executed completely, regardless successfully or failed, the TriggerEventHandler instance will trigger the jobs which has a dependence on this job according the event it raised.

Here is the pseudo code:

```
if (!event.equals(EventHandler.NOTSTARTED)) {
   // get all the dependent jobs and start the jobs if the dependence matches.
   List jobs = processor.scheduler.getAllDependentJobs(job);
   for each depJob in jobs {
      if (depJob.getDependence().getDependentEvent().equals("BOTH") ||
         depJob.getDependence().getDependentEvent().equals(event)) {
             // set up the start/end time of the job to ensure the
      job starts to
            // execute in the depJob.getDependence().getDelay()
      milliseconds and
            // execute only once.
            Job sjob = new Job(depJob);
            int delay = depJob.getDependence().getDelay();

            GregorianCalendar startTime = new GregorianCalendar();

            startTime.add(Calendar.MILLISECOND, delay);

            sjob.setStartDate(startTime);

            //calculates how many milliseconds have elapsed since
      start of today
            GregorianCalendar startOfToday = (GregorianCalendar)
      startTime.clone();
            startOfToday.set(Calendar.HOUR_OF_DAY, 0);
            startOfToday.set(Calendar.MINUTE, 0);
            startOfToday.set(Calendar.SECOND, 0);
            startOfToday.set(Calendar.MILLISECOND, 0);
            sjob.setStartTime((int) (startTime.getTimeInMillis() -
      startOfToday.getTimeInMillis()));

            //delay will never be more than 24 hours, this will ensure
      that stopDate will in fact not
            //occur before the job is to start! And also, it makes
      sure the job is only run once.
            GregorianCalendar stopDate = (GregorianCalendar)
      startTime.clone();
            stopDate.add(Calendar.MILLISECOND, ONE_DAY);
            sjob.setStopDate(stopDate);
```

```
                    sjob.setIntervalUnit(new Year());
                    sjob.setIntervalValue(1);
                    sjob.setRecurrence(1);

                processor.schedule(sjob);
            } else {
                // if the depJob is not triggered to started, the NOTSTARTED
                // event will be raised.
                depJob.fireEvent(EventHandler.NOT_STARTED);
            }
        }
    }
```

1.3.5

Logging

Logging is performed in two places in this component: JobProcessor and TriggerEventHandler. This section serves as an authoritative list of where and how logging should be performed, in lieu of putting this information in method documentation.

The entry and exit of all methods will be logged at TRACE level. Any exception is logged at ERROR level.

1.4 Component Class Overview

JobProcessor

This class used to retrieve and run scheduled jobs that have specified scheduled time. It will maintain two running tasks. One for reloading jobs from the scheduler, and another for executing them. The current and reloaded jobs will be held in one staging list, whereas executing jobs, or rather copies of them, will be held in another.

TriggerEventHandler

It used to start the jobs that are dependent on the completion of the job. The jobs can be configured to be dependent on the success or failure or both of another job.

1.5 Component Exception Definitions

JobProcessorException

This exception indicates a failure start or shutdown of the processor, but not used currently in this component.

1.6 Thread Safety

This component is thread-safe. JobProcessor and TriggerEventHandler are immutable and thread-safe.

2. Environment Requirements

2.1 Environment

. Development language: Java1.4
Compile target: Java1.4, Java1.5

2.2 TopCoder Software Components:

Job Scheduler 3.0

This is the other half of the previous component, and it now contains the Scheduler interface as well as the Job and so forth.

Base Exception 1.0

Provides a uniform base exception class.

Logging Wrapper 1.2

Used by JobProcessor and TriggerEventHandler to log.

2.3 Third Party Components:

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.scheduler.processor

3.2 Configuration Parameters

None

3.3 Dependencies Configuration

None

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

None

4.3 Demo

```
        ConfigManager.getInstance().add("log.xml");

        //here is a predefined Scheduler
        Scheduler scheduler = new MyScheduler();
        //we can add a job to the scheduler so that the processor can load
and execute it
        scheduler.addJob(createImmediateJob());

        //defines the reload interval
        int reloadInterval = 15 * 60 * 1000;

        //defines a logger
        Log log = LogFactory.getLog();

        //creates a processor
        JobProcessor processor = new JobProcessor(scheduler, reloadInterval,
log);

        //starts the processor, jobs from scheduler can be automatically
scheduled
        processor.start();

        //we can also get those jobs on execution
        List executingJobs = processor.getExecutingJobs();

        //now stop the job
        processor.stopJob("test");

        //and shut down the processor
```

```
    processor.shutdown();
```

5. Future Enhancements

None
.