Job Scheduler 2.0 Component Specification

1. Design

A job is a specific O/S Command/java class to be launched on a particular schedule. Job information is stored in a configuration file. Jobs can be created either through this component or through manual edit of the configuration file.

A job can be of two specific types. The first, External, are for operating system executables and scripts. This component uses the com.topcoder.util.exec package to execute these. The second, Internal, are Java classes implementing both the java.lang.Runnable and com.topcoder.util.scheduler.Schedulable interfaces. The component uses reflection to load and execute these classes.

Support for multiple schedules is provided by re-adding a job with a different name and a new schedule.

Both internal and external jobs are executed asynchronously. This permits simultaneous and overlapping jobs.

The com.topcoder.util.log component is used to log the results (run start, run completion, status) of each launched job. This provides an essential audit trail of all execution attempts and results.

Version 2.0

Additional Functions

1. Job Dependency

A job can be dependent on another job regarding execution time. For example Job B does not have a configured date time. It is configured to run on the successful completion of Job A. Allow the option to execute Job B even if Job A fails. Also allow configuration of an option time delay before Job B executes after Job A. However, a Job must have either a scheduled date time or a task dependency but not both.

2. Alert Notification

Allow set up of an email alert if a job returns a failure or cannot be executed. Configuration is at the job or job group level.

3. Job Group

A job group is a grouping of jobs to simplify job scheduler configuration. A group can contains one or more jobs. A job can belong to one or more groups.

In current version, the job groups are used to ease “Alert Notification Configuration”.

Design Details:

1. ScheduledEnable Interface

ScheduledEnable extends Schedulable and Runnable interfaces. An internal job (Java class) using the functions in version 2.0 must implement this interface.

The additional methods to implement are used to get the running status and returned message data of the job.

2. **EventHandler Interface**
   There are three event of a job: Not Started, Executed Successful, and Executed Fail. The EventHandler is designed to handle these events of jobs.

   EmailEventHandler – Used to send email alert notifications when a job returns a failure, success or cannot be executed.

   TriggerEventHandler – Used to start the jobs which are dependent on the completion of the job. It can be configured to be dependent on the success or failure or both.

3. **JobProcessor**
   This class is abstracted from the Scheduler in version 1.0. It is used to schedule jobs which have specified scheduled time.

1.1 Reference any design patterns used

Listener Pattern – EventHandler use the listener pattern. It is used to send email alert or trigger the dependent job to start in this version.

1.2 Reference any standards used in the design

None.

1.3 Explain any required algorithms for the implementation (provide pseudo code)

1. Scheduler Algorithm:

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

Implementation:
The scheduler algorithm is implemented in the JobProcessor class.
When new a JobProcessor will start a heartbeat thread to schedule the jobs in the processor. 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.

```
timer = new Timer();
timer.schedule(
    new TimerTask(){
        public void run() {
            GregorianCalendar rightNow = new GregorianCalendar();
            synchronized (jobs) {
                for (int i = jobs.size() - 1; i >= 0; i--) {
                    Job job = (Job) jobs.get(i);

                    if (job.getStop() != null &&
                         (rightNow.after(job.getStop()) ||
                         job.getNextRun().after(job.getStop())) {

                        // the job is expired, remove it.
                        jobs.remove(i);
                    }
                    else if (rightNow.after(job.getNextRun()) {
                        // the job is ready to start, first set the
                        // lastRun and nextRun time, then start it.
                        job.setLastRun(rightNow);
```

```
                    GregorianCalendar nextRun = new
                    GregorianCalendar();
                    nextRun.setTimeInMills(rightNow.getTimeInMillis());
                    nextRun.add(job.getIntervalUnit(),
                         job.getIntervalValue());

                    job.setNextRun(nextRun);

                    Job executingJob = new Job(Job);
                    executing.add(executingJob);
                    executingJob.start();
                }

            }
        }

        synchronized (executing) {
            for each job in executing list {
                if (job.isDone()) {
                    if (job is external or internal but have
                        ScheduledEnable interface implemented) {

                        job.fireEvent(job.getRunningStatus());
                    }
                    remove job from the executing list.
                }
            }
        }
    }
}, 0, 1000);
```

2. 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 scheduler.

The basic algorithm is:

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

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

3. Schedulable and Runnable

Internal java classes are loaded using reflection. To ensure a consistent interface, those classes need to implement both Schedulable and Runnable. Schedulable contains a Stop() method which, when called, should terminate the object's thread and execution. To start the class instance, a call to the Runnable's run method, in a new Thread, will be made.

4. Schedule Entry – Add, Modify, Delete.
These methods should work directly on the data in the configuration file. After the data in the configuration file has been altered, any necessary modifications to the data in the schedule list are done. For example, if a job had previously not been loaded as it's last run date is in the past, and a modify call extends that date into the future, then, after the modifications are stored in the configuration file, the job should be loaded into the scheduler. As another example, if a job is deleted, then after removing the data from the configuration file, the job should be removed from the scheduler list. Note that if the job is currently executing, it should be left in the execution list unless a specific stop command is called against it.

If adding a job that already exists (based solely on job name), or attempting to modify or delete a non-existent job, a jobActionException will be thrown.

A manual modification of the configuration file will not take effect until the next time the scheduler is either shut down and restarted, or stopped and started. Note that a manual modification may be overwritten by an API update if the manual modification is made while the scheduler is running. Generally speaking, manual modification is not recommended during scheduler execution.

5. Send Email Alert
Purpose: To send email when scheduled job has an event.

Implementation:
When a job executed successful, failed or even not started, the job will raise the corresponding event. The EmailEventHanlders are listened to the job and send email alert according the event.

Here is the pseudo code:

```
if (event.equals(requiredEvent)) {
    //1. Generate the message
    NodeList msgdata = job.getMessageData();
    Node[] nodes;
    if (msgdata == null) nodes = new Node[2];
    else nodes = new Node[msgdata.getNodes().size() + 2];

    nodes[0] = new Field("JobName", job.getName(), null, true);
    nodes[1] = new Filed("JobStatus", event, null, true);
    copy the nodes in msgdata.getNodes() to the rest of the array.

    TemplateFields data = new TemplateFields(nodes, template);
    String message = DocumentGenerator.getInstance().applyTemplate(data);

    // 2. send the email using EmailEngine.

    TCSEmailMessage email = new TCSEmailMessage();
    email.setSubject(job.getScheduler().getEmailAlertSubject());
    email.setFromAddress(job.getScheduler().getEmailFromAddress());
    for each recipient in recipients {
            email.addToAddress(recipient, 0);
    }
    EmailEngine.send(email);

}
```

6. Trigger the job which has a dependency job to start

   Purpose:

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

   Implementation:

   This function is implemented using the TriggerEventHandler. When a job is created and added to scheduler, TriggereEventHandler will be always added 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 = job.getScheduler().getAllDependentJobs(job);
    JobProcessor processor = job.getScheduler().getJobProcessor();
    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() ms and
                    // execute only once.

                    Job sjob = new Job(depJob);
```

```
                GregorianCalendar startTime = new GregorianCalendar();
                GregorianCalendar endTime = new GregorianCalendar();
                startTime.add(Calendar.MILLISECOND,
                        depJob.getDependence().getDelay());
                endTime.add(Calendar.MILLISECOND,
                        depJob.getDependence().getDelay() + 10000);

                sjob.setStartTime(startTime);
                sjob.setEndTime(endTime);
                sjob.setIntervalUnit(Calendar.YEAR);
                sjob.setIntervalValue(1);

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

1.4 Component Class Overview

Scheduler:

This class is the major class of Job Scheduler component.It manages jobs and job groups and has an job processor to schedule jobs.

Job

This class is the specific job instance to schedule. A Job must have either a scheduled date time or a task dependency but not both.

JobGroup

A job group is a grouping of jobs to simplify job scheduler configuration. A group can contains one or more jobs. A job can belong to one or more groups.In current version, the job groups are used to ease “Alert Notification Configuration".

Dependence

A job can be dependent on another job regarding execution time. This class represents this relationship. It has a dependent job name, dependent event and delay properties.

JobProcessor

This class is abstracted from the Scheduler in version 1.0. It is used to schedule jobs which have specified scheduled time.

EventHandler

There are three event of a job: Not Started, Executed Successful, and Executed Failed. The EventHandler is designed to handle these events of jobs.

TriggerEventHandler

It used to start the jobs which 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.

EmailEventHandler

It is used to send email alert notifications when a job raises an event matching the requiredEvent initialized in ctor. The email message is generated by Document Generator component.

ScheduledEnable

The ScheduledEnable interface is the only required interface for classes run as a job in Job Scheduler 2.0. It extends Schedulable interface(v1.0) and Runnable interface, And forces the implemention classes return running status and message data at runtime..

Schedulable

The Schedulabe interface is one of two required interfaces (the other being java.lang.Runnable) for classes to be run as a job.

1.5 Component Exception Definitions

JobActionException [Custom]

This exception will be thrown when the configuration file is invalid or the job is failed to schedule.

IllegalArgumentException

This exception is thrown in various methods where the parameters in methods are invalid.

IllegalStateException

This exception is thrown in various methods where the user access an object when it is the illegal state.

1.6 Data Item Definitions

1. Job Data Item description:

a. Job start date/time: The date/time on which a job is first scheduled to run.
b. Job stop date/time: The date/time after which the job should no longer run.
c. Job run interval: The time interval when the job should run.
d. Job Name: unique name identifying this job.

Example:
Job start date/time: 01 March at 0315.
Job stop date/time: 01 March at 0730.
Job run interval: 1 hour.
Based on the above, and assuming the component is active prior to 01 March and continuously through 02 March, the job would run at the following times:

0315 01 March
0415 01 March
0515 01 March
0615 01 March
0715 01 March

Note that the Job stop date/time indicates the last time the job should be scheduled for, and does not mean the job should be terminated if already executing. In the above example, if the job takes 30 minutes to run, it would run past the job stop date/time, which is fine. The job itself won't be rescheduled again. Also, if the job stop date/time is null, that indicates there is no 'stop' time for that job and it should continue to be rescheduled indefinitely (However, a non-null, valid start date/time is required).

2. Interval value and unit, and start / stop date time.

Internally the date and time is stored using a java.util.GregorianCalendar object. Intervals are stored as two parts: an integer value, and an integer unit which maps to the java.util.Calendar types. The allowed unit types are:

java.util.Calendar.YEAR	Indicates year intervals.
java.util.Calendar.MONTH	Indicates month intervals.
java.util.Calendar.DATE	Indicates day intervals.
java.util.Calendar.WEEK_OF_YEAR	Indicates week intervals.
java.util.Calendar.HOUR	Indicates an hour interval.
java.util.Calendar.MINUTE	Indicates a minute interval.
java.util.Calendar.SECOND	Indicates a second interval.

3. Configuration File Data Items

The Property Name Space is the job name.
Within each property name space the following name:value pairs are stored:
StartDTG: The start date / time for the job.
EndDTG: The end date/time for the job.
IntervalValue: The numeric interval value for the job.
IntervalUnit: The unit of time (see #4 above) for the interval value.
JobType: Whether the job is an external command or internal java class.
JobCmd: The job execution string (either external command or java class name, depending on the JobType value).

4. Log File Data Items

For each job executed (either completed, stopped, or unable to launch), the following information will be written out to the log file:
Job Name : launch date and time : stop date and time: scheduler result: job status

The status is the String text, if any, returned from the executing object.
The scheduler status is whether the job was terminated normally (execution completed) or abnormally (close() or shutdown() was invoked or launch failed or error occurred).

5. Invalid Jobs
The following conditions would make a job invalid during creation or modification:
   a. A start date past the end date.
   b. A name which already exists.
   c. A name which is null or an empty String.
   d. A negative or 0 increment value.

e. An increment unit value not following one of the Calendar types outlined in Section 1.6.2.
f. A job type not one specified in the Scheduler Object.
g. A null start date.

1.7 Thread-Safe

This component is thread-safe. Scheduler, JobProcessor, JobGroup, Dependence, and all EventHandler classes are thread-safe. The only exception is Job class, same properties in it are mutable, it is designed that you can't modify those mutable properties. The usage of Job in this component is thread-safe.

Also, the implementations of the Schedulable or ScheduledEnable interfaces should be thread-safe, that means when the job is running, the isDone, getRunningStatus, etc methods can be called in thread-safe way.

2. Environment Requirements

2.1 Environment

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

2.2 TopCoder Software Components:

Configuration Manager 2.4

Logging Wrapper 1.2

Executable Wrapper 1.0

Document Generator 2.0

Used to generate the email message.

Email Engine 3.0

Used to send the email.

Note: The Alert Factory and Event Email Processor are not used. Because both of them need to persist the messages and are a little complex. In this component, it is not worth to add this complexity while using these two components.

2.3 Third Party Components:

None.

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.scheduler

3.2 Configuration Parameters

Parameter	Description	Values
CMConfig.<JobName>	Define a job	Can be any string except "DefinedGroups" and "EmailConfiguration"
CMConfig.<JobName>.Value	Basic parameter of the job If the the job is not dependent on another job, StartDTG, InternalValue, InternalUnit, JobType, and JobCmd are required, otherwise, only JobType and JobCmd is required.	Begin with the param name. I.E StartDTG: JobType: etc
CMConfig.<JobName>.Dependence	Specify the depended job, Optional	
CMConfig.<JobName>.Dependence.<dependedJobName>	The name of the depended job	Must have been defined in the configuration file
CMConfig.<JobName>.Dependence.<dependedJobName>.Status	Tell when the job should be started, on which status of the completion of the depended job, required	One of SUCCESSFUL, FAILED or BOTH
CMConfig.<JobName>.Dependence.<dependedJobName>.Delay	Specify a time delay before the job is triggered, optional.	Non-negative value, the unit is ms.
CMConfig.<JobName>.Messages	Define the message alerts of this job, Optional	
CMConfig.<JobName>.Messages.<Status>	When to send the email alert, optional	One of SUCCESSFUL, FAILED and NOTSTARTED
CMConfig.<JobName>.Messages.<Status>.TemplateFileName	The name of the template file to generate message, optional. If it is not specified, the default template in scheduler will be used.	File name
CMConfig.<JobName>.Messages.<Status>.Recipients	The Recipients of the message alerts, required	Email addresses
CMConfig. DefinedGroups	Define job groups, optional	
CMConfig. DefinedGroups.<GroupName>	The job group name, optional	
CMConfig. DefinedGroups.<GroupName>.Jobs	The jobs this group contained, required, at least one.	The name of defined jobs
CMConfig. DefinedGroups.<GroupName>.Messages	The message alerts configuration based on the group, similar with configuration on a single job. Optional	Similar with the configuration on a single job.
CMConfig. EmailConfiguration.FromAddress	The from address in the alert messages in this component, required	Valid email address
CMConfig. EmailConfiguration.S	The title of the email alert in this component, required	string

ubject		
CMConfig. EmailConfiguration.DefaultTemplate	The default template file of the email message. It is used when the template of a specific email alert is not set in the configuration file. Required.	Valid template file. See Document Generator component.

3.3 Dependencies Configuration

See the sample_config.xml

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

Follow demo.

4.3 Demo

4.3.1 Functions in version 1.0

```
//Instantiate the Scheduler, passing it the name of the config file
// containing job data.
Scheduler myScheduler  = new Scheduler ("myScheduler");

//For first time runs, add new jobs.
//This job will start at 1 am on the 10th of March (GregorianCalendar
// months
//  run from 0 to 11), and will run once a day, at 1 am, everyday until
//  the 10th of March 2004 (inclusive).
Job deleteFiles = new Job("Nightly file cleanup",
    new GregorianCalendar(2003, 04, 10, 01, 00, 00),
    new GregorianCalendar(2004, 04, 10, 01, 00, 00),
    1,
                        Calendar.DATE,
     myScheduler.JOB_TYPE_EXTERNAL,
                         "erase *.tmp");
myScheduler.addJob(deleteFiles);

//Start the scheduler.
myScheduler.start();

…..
// At some later point, after the job has launched,
// check status of executing jobs
ArrayList status = getJobExecutionList();
For (int j = 0; j<status.size(); j++)
      String status = ((Job)status.get(j)).getStatus();

//Similarly, check on the jobs in the schedule list.
ArrayList schedStatus = getJobList();
For (int k=0; k<status.size(); k++){
     //it is deprecated in version 2.0
```

```
        String status = ((Job)schedStatus.get(k)).getStatus();
        String runningStatus = ((Job)schedStatus.get(k)).getRunningStatus();

        GregorianCalendar lastRun = ((Job)schedStatus.get(k)).getLastRun();
        GregorianCalendar nextRun = ((Job)schedStatus.get(k)).getNextRun();
    }
```

4.3.2 Functions in version 2.0

```
    // Add a job dependent on another job regarding the execution time.
    // jobA has a specific schedule time. jobB is dependent on jobA and
    // jobC is dependent on jobB
    Job jobA = new Job("jobA", startTime, stopTime, 1, Calendar.DATE, 1,
          "com.topcoer.MyJob");

    myScheduler.addJob(jobA);

    Job jobB = new Job("jobB", 1, "com.topcoder.MyJob2",
          new Dependence("jobA", EventHandler.SUCCESSFUL, 10000);
          // the delay is 10s

    myScheduler.addJob(jobB);

    Job jobC = new Job("jobC", 0, "dir",
          new Dependence("jobB", EventHandler.SUCCESSFUL, 0);
          // no delay

    // add email alert event handler to jobC, if the jobC executed
    // unsuccessfully, an email will be sent to name1@topcoder.com
    // the typical messageTemplate is like
    //
    // The Job %JobName% is %JobStatus%...
    //
    jobC.addHandler(new EmailEventHandler(messageTemplate,
          Collections.asList(new String[]{"name1@topcoder.com"}),
          EventHandler.FAILED);

    myScheduler.addJob(jobC);

    // add a job group to scheduler
    JobGroup group = new JobGroup("group_1",
          Collections.asList(new Job[]{jobA, jobB, jobC}));

    myScheduler.addGroup(group);

    // add an email Event Handler to the group
    // the following code means if any one of the jobs in the group executed
    // successfully, an email alert will be sent to "name2@topcoder.com" and
    // "name3@topcoder.com"
    group.addHandler(new EmailEventHandler(messageTemplate,
          Collections.asList(
                new String[]{"name2@topcoder.com", "name3@topcoder.com" }),
          EventHandler.SUCCESSFUL);


    // you can remove the handler from job and group at runtime.
    jobC.removeHandler((EventHandler) jobC.getHandlers.get(0));
    group.removeHandler((EventHandler) group.getHandlers.get(0));

    // you also can remove job and group from the scheduler
    myScheduler.removeGroup(group);
```

```
myScheduler.removeJob(jobA);
```

5. Future Enhancements

Add more EventHandlers to provide more functions in Job Scheduler.

.