[TopCoder Software]

Component Specification

1. Design

The Executable Wrapper component facilitates correct execution of commands in the operating system outside the JVM. Java already supports this through java.lang.Runtime and related classes, but incorrect use can cause undesired results or even deadlock.

Safety
This component exposes a "safe" version of Java's Runtime API, in that it correctly uses the java.lang.Runtime API to create a Process and wait for its completion. It avoids the one big trap in the Runtime API - not reading the standard output and standard error streams while waiting for completion - which can cause deadlock. Separate threads are used to consume the output of these two streams.

The component also shields users from details of the operating system's shell and command execution facilities. Some commands (like "time" in Unix shells, or "dir" in the Windows shell) are not executable programs but are interpreted by the shell itself. Likewise some familiar syntax (like "~" as a synonym for the user's home directory) is actually only interpreted by the shell. This component invokes commands using a shell appropriate for the current operating system, so that these features are available, and work as is generally expected by the user.

Additional functionality
The Executable Wrapper augments the Runtime API by providing complete implementations for common usage scenarios: synchronous or asynchronous execution, with or without a timeout. The Runtime API itself only supports synchronous execution without a timeout.

The implementations of Executor in this component support these various kinds of command executions, and form the core of this component. These objects execute a process and subsequently return the process's exit status as well as its standard output and error output as Strings, for convenience (in an ExecutionResult object).

Finally, for asynchronous execution of commands, the component returns a "handle" to the executing command (AsynchronousExecutionHandle) that can be used to check its status, get the result, or halt the command.

Future extensibility
Internally, the component encapsulates all of its platform-specific functionality in subclasses of PlatformSupport. PlatformSupport has a factory method that will retrieve a subclass appropriate for the OS in which the JVM is currently executing. Currently this is used to create a command that uses a shell appropriate to the current operating system. This can also support future extensions of the Executable Wrapper that utilize more platform-specific code (JNI) to provide functionality beyond what is possible with Runtime.

The component is designed such that support for additional platforms can be added dynamically.

1.1 Reference any design patterns used

The PlatformSupport class is a Singleton; it also exhibits the Factory pattern because its "getInstance" method selectively instantiates a subclass of itself according to the current operating system.

The implementations of Executor (SynchronousExecutor, TimeoutExecutor, and AsynchronousExecutor) make good use of the Decorator pattern. The TimeoutExecutor extends the functionality of SynchronousExecutor, not by subclassing it, but by enclosing it. Likewise the AsynchronousExecutor encloses a TimeoutExecutor or SynchronousExecutor to do its work.

Finally, Exec is a Facade that presents the simplified public interface to the component.

1.2 Reference any standards used in the design

None.

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

The following article should be useful during implementation:
http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-traps.html

This page will also be useful during implementation:
http://www.tolstoy.com/samizdat/sysprops.html
It lists the value of the system property "os.name" for various operating systems; this is needed in PlatformSupport to identify the current OS and change command execution to suit the current OS.

See the javadoc for the PlatformSupport class and its subclasses for further comments on this.

The sequence diagrams should receive special attention in their illustration of the execute() methods. They show in detail exactly how this should be implemented. The design attempts to stress that the SynchronousExecutor is the only place where the Runtime API needs to be used; the TimeoutExecutor and AsynchronousExecutor "decorate" this class in order to provide more sophisticated functionality.

Note: the skeleton source code under java/main has javadoc with further comments and specifications for implementation.

1.4 Component Exception Definitions

IllegalArgumentException:
An IllegalArgumentException is thrown from Exec's execute() methods when:

- A "command" argument is null, or empty (length 0)
- A "parameters" argument is null
- A "timeout" argument is not positive (0 is not allowed)

IllegalArgumentException is also thrown when PlatformSupport.registerPlatformSupport() is called with a null "osString", or "platformSupport" object.

IllegalStateException:
An IllegalStateException is thrown when getExecutionResult() or getExecutionException() is called on AsynchronousExecutionHandle before the command that it is a handle to has finished.

ExecutionException:

An ExecutionException is thrown from Executor.execute() when any exception occurs while actually executing the command (exceptions from the Runtime and Process classes). These should be caught and re-thrown as an ExecutionException with the same message.

This is only actually thrown in SynchronousExecutor, but since this class is used by TimeoutExecutor, it will receive and rethrow this Exception as well from its execute() method. In the case of AsynchronousExecutor, its execute() method does not actually throw an ExecutionException since it returns immediately, but any Exception that occurs will eventually be made available in the AsynchronousExecutionHandle that it returns to the caller.

ExecutionTimedOutException:
TimeoutExecutor throws an ExecutionTimedOutException if the command that it is executing does not complete within the given timeout period.

ExecutionHaltedException:
If halt() is called on an AsynchronousExecutionHandle, it will cause an ExecutionHaltedException to be thrown from the underlying SynchronousExecutor. So, shortly after halt() is called on an AsynchronousExecutionHandle, its getExecutionException() method will return an ExecutionHaltedException.

2. Environment Requirements

2.1 TopCoder Software Components:

None.

2.2 Third Party Components:

None.

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.exec

3.2 Configuration Parameters

None.

3.3 Dependencies Configuration

None.

4. Usage Notes

4.1 Required steps to use the component

Simply call the static methods of Exec; here we list all .java files in the current directory and subdirectories:

```
ExecutionResult result =
      Exec.execute(new String[] {"ls", "-R", "*.java"});
System.out.println(result.getExitStatus());
System.out.println(result.getOut());
System.out.println(result.getErr());
```

This will block until the command completes.

This also causes the command to be executed with the current process's environment and working directory. If the caller wants to use different values, the "full" version of this or any other execute() or executeAsynchronously() method can be used, which takes an ExecutionParameters object. This can be used to specify the command's environment variables or working directory:

```
ExecutionParameters parameters =
      new  ExecutionParameters(new String[] {"ls", "-R",
      "*.java"});
Map environment = new HashMap();
environment.put("myProperty", "myValue");
parameters.setEnvironment(environment);
File workingDir = new File("/tmp");
parameters.setWorkingDirectory(workingDir);
ExecutionResult result = Exec.execute(parameters);
...
```

One can also specify a timeout:

```
ExecutionResult result =
      Exec.execute(new String[] {"ls", "-R", "*.java"},
      500);
```

This will block for at most 500ms (half a second); if the command has not completed in 500ms it is forcibly terminated and the method returns.

Finally, one can also execute a command asynchronously:

```
AsynchronousExecutionHandle handle =
      Exec.executeAsynchronously({"cvs", "update"});
```

The method returns immediately, and the returned handle can be used to check the status of the command, or end it:

```
boolean willingToWait = true;
while(!handle.isDone()) {
      if(!willingToWait) {
            handle.halt();
            break;
      }
      .. wait, do stuff ..
}

if(handle.getExecutionException() != null) {
      // handle the exception
}
ExecutionResult result = handle.getExecutionResult();
if(result.getExitStatus != 0) {
      // error in cvs
}
...
```

Once the command has finished, isDone() will return true. Then the execution result is available by calling getExecutionResult(), unless an exception occurred while executing, in which case this returns null but getExecutionException() returns the thrown exception.

Finally, commands can be executed asynchronously with a timeout; the command will be automatically terminated if it does not complete within a given amount of time.

4.2 Demo

This code retrieves load average information from the machine, which might be useful in a management application.

```
public double getLoadAverage() {
      ExecutionResult result = null;
      try {
            result = Exec.execute(new String[] {"uptime"});
      } catch(ExecutionException ee) {
            return Double.NaN;
      }
      if(result.getExitStatus() != 0) {
            String out = result.getOut();
            // out is like "10:47PM  up 13:51, 2 users, load
            averages: 1.12, 1.17, 1.20"
            int start = out.indexOf("load averages: ") + "load
            averages: ".length();
            int end = out.indexOf(",", start);
            return Double.parseDouble(out.substring(start, end));
      } else {
            return Double.NaN;
      }
}
```

As always, this will only work on operating systems that have an "uptime" command.

This code kicks off a long remote copy:

```
public void copyToBackup() {
      Exec.executeAsynchronously(new String[] {"scp", "-r",
      "/data", "user@backup:/backup/data"});
      // returns immediately
}
```

In this scenario, the handle to asynchronously executing process isn't used.

Or, if it's not desirable to let the copy run for more than, say, 6 hours, one can also specify a timeout:

```
public void copyToBackup() {
      Exec.executeAsynchronously(new String[] {"scp", "-r",
      "/data", "user@backup:/backup/data"}, 6*60*60*1000);
      // returns immediately - process will be killed if it runs
      for more than 6 hours
}
```