Tool interface

Custom Tool is written in the Java language. The tool implements a specialized interface called ITool. The Report API provides both an interface and a class to support interface realization and class generalization. As mentioned earlier, a Tool is modeled on the JavaBean specifications. Functions implemented in this class must be defined in a series of setter or getter methods. The following sample shows the source code for the ITool interface that you must implement:

```
public interface ITool extends IBean
{
   // attributes
   Void VOID = new Void();
   // inner classes
   public class Void{}
   public class RetainedString implements CharSequence{}
   // methods
   void setContext(IContext context);
   String getContext();
   void setProperties(Properties properties);
   Properties getProperties();
}
```

All tools must implement the ITool interface (or one of its subclasses) as it defines the methods the template engine calls to execute. The table below provides a description of the methods in the ITool interface.

The table below lists Description of Methods in ITool interface.

Method	Description
setContext(Context context)	Once the class has been initialized, this method is called by the template engine runtime to set the template context. Overriding the current context will affect the code after this tool.
getContext()	Return the template context that is assigned to the current runtime.
setProperties(Properties properties)	This method is called by the template engine runtime, once the class has been initialized, to set the template properties. Overriding the current properties will not affect the engine.
getProperties()	Return the current template properties.
VOID	Represents the Void class. VOID is used to make sure that returning data from the setter method is absolutely nothing. In general, Velocity considers the return of void from the setter method as a ‘null’ value. This causes the word 'null' to be printed out in the report. To avoid this problem, the setter method may use VOID as a return object.
class Void	A void class is used as a return in the Tool when you want to be sure that nothing is returned to the context.
class RetainedString	A direct command report formatter to keep the referenced String format. The formatter and other reference insertion handlers should maintain Strings directed by this class.

The other classes that can be extended are Tool and ConcurrentTool.

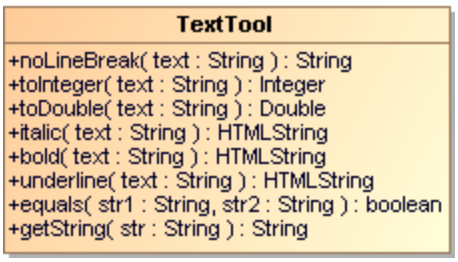

Figure 1: ITool Interface And Related Class.

Class Tool

A Class Tool is a base class realizing the interface ITool. This class provides the default implementation for ITool.

This class also provides methods derived from the java.util.Observable class, which can be used to notify the observers. An observer class, such as a template engine or a graphical user interface class, can receive the notification message from the tool and manage to display or interact with the user.

Concurrent Tool

A Concurrent Tool provides concurrent tasks running in the template engine. This class implements an unbounded thread-safe queue, which arranges the elements in a first-in-first-out (FIFO) order. New tasks are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.

The tool extends from this class and is ideal for processing a long task that does not want other tasks to wait until the process is complete. The following code shows the sample usage of Concurrent Tool.

```
import com.nomagic.magicreport.engine.ConcurrentTool;
public class LongTaskTool extends ConcurrentTool
{
   public String longTask() {
      // enqueue object
      offer(new ConsumeObject(referent));
   }
   public void consume(ConsumeObject consumeObject) {
      Object referent = consumeObject.get();
      // process long task
   }
}
```

An example of Concurrent Tool is *FileTool*. The root template that calls the **file tool** method will create a subprocess, offer the subprocess into a queue, and continue the root template until finished. The template engine will later call the **consume** method to complete the subprocess once the consumer queue is available. The number of concurrent processes available for the execution is declared in the template engine property (See **TemplateConstants.TEMPLATE_POOL_SIZE** for detail.).