Chapter

14

14 API Concepts and Usage

SpectrumSCM is a process driven Source Configuration Management System that can be used to manage the life cycle of any electronic asset. Users of the system define workflows in the tool that correspond to the processes that are already in place within their organizations. By default, workflows defined in SpectrumSCM are considered ad-hoc, which means that work items can be assigned from any user defined phase into any other user defined phase[2]. The responsibility of moving work items from one phase to another falls on the shoulders of users that have been assigned that particular responsibility[3] (Project Managers, etc…).

With the advent of the graphical workflow system under release 2.4, Spectrum has introduced the capability to perform some automations directly through the UI. These include the "Promotion Recipient" and callouts. The API mechanism is still supported as a general way to extend the SpectrumSCM system and in particular, because it is Java based instead of shell based, it is more efficient.

The purpose of the SpectrumSCM API (Application Programming Interface) is to allow users of the system to construct automated business systems by implementing a well defined set of event triggers and interfaces. An automated business system relieves the burden of making particular users or team leaders responsible for performing certain tasks manually and adds the ability to automate decision making processes, including the automation of interfaces to external systems. External system integration allows issue tracking numbers and content from external systems to be easily integrated into and out of the SpectrumSCM system.

Custom programs that implement the SpectrumSCM APIs are written in the Java programming language and are known as *plugins* in SpectrumSCM. *Plugins* are easily compiled and can easily be added to a running SpectrumSCM server through an XML interface. Plugins are loaded dynamically into the server at run time. Dynamic loading also allows the plugins to be changed by the developer and reloaded without impacting the server. The SpectrumSCM plugin

---

[2] Only users with the proper permissions model can actually assign work items. Such ad-hoc transitions are still fully recorded to provide the required audit trail.

[3] See the SpectrumSCM User Guide Chapter 5 on User Management.

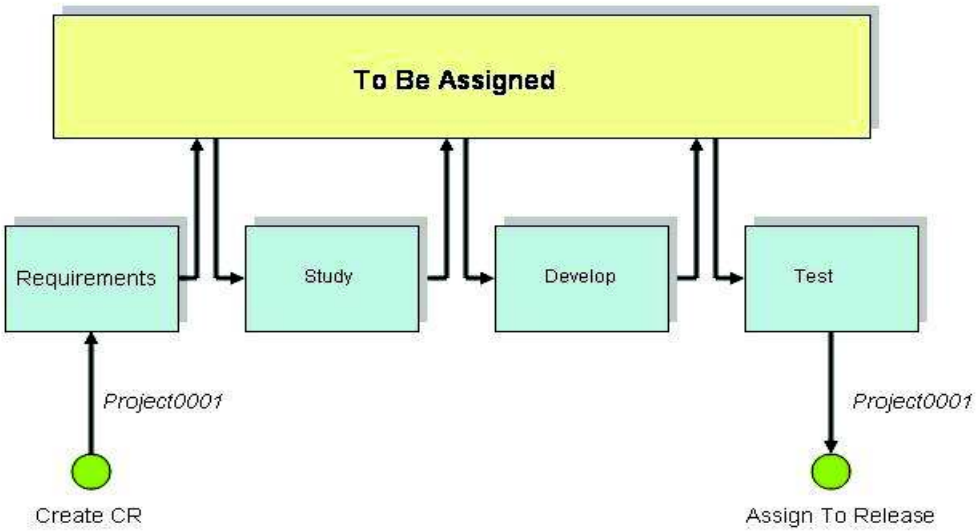
configuration allows for one or more plugins to be actively configured into the system. Individual plugins can be turned on or off by simply modifying the plugins XML definition file.

14.1 Manual vs. Automated Workflow

By default, the SpectrumSCM system provides for the manual ad-hoc workflow process. This decision was made because of the complexities involved with trying to configure and support an automated workflow system right out of the box. The manual system allows the majority of users to quickly and easily get started with SpectrumSCM. The automated systems, both in terms of the graphical workflow and the API, allow the more advanced environments the flexibility they need to configure/automate their needs. Every organization has a different definition of what a workflow system should accomplish and how it should work. Our goal was to be able to handle the common denominator of all workflow systems and then allow for complete customization through server extensions. SpectrumSCM has accomplished this goal with their base system, the graphical workflow and the API extensions.

14.2 A Typical Simple Workflow Process

The SpectrumSCM system allows the user to create completely customizable workflows. Through the use of API mechanism, users can tailor their business process workflow to be cognizant of overall business practices. This allows for external business rules to be executed as part of the transition of an issue or Change request. The following workflow diagram is an example of a trivial software development workflow.

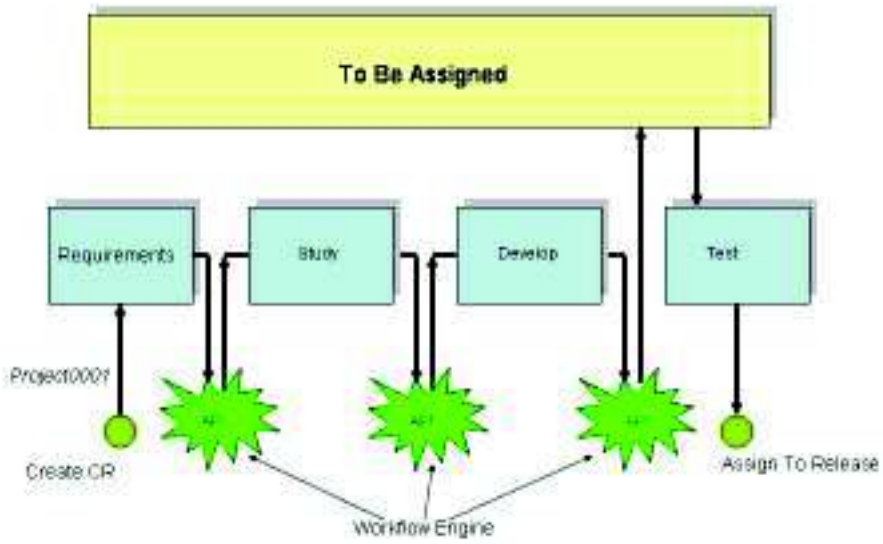


In this example, there are four user defined life-cycle phases *Requirements*, *Study*, *Develop* and *Test*. In the SpectrumSCM system, Change Requests (CRs) can be created and immediately assigned to any phase within the defined life-cycle. In this example, the CR (*Project0001*) was initially created and assigned to the *Requirements* phase. As Change Requests are manually

progressed from phase to phase by the end user, they are not immediately progressed into the next defined phase. Each Change Request is first progressed into the **TBA**[4] super phase. The TBA super phase is where all manually progressed change requests are assigned, unless they are directly assigned into another user defined state by a user with the proper permissions. This supports the working model where a developer or tester does not have assignment privileges, but the project/team lead does. The appropriate user (project/team lead in the previous example) would then manually assign the CR into the next appropriate phase. When change requests are promoted into the TBA phase, e-mail notifications are sent to the appropriate users so that process decisions about where the CR should go next can be made.

14.3 An Automated Workflow

In an automated workflow, progression decisions for manually progressed Change Requests are delegated to an automated workflow engine. The engine, in this case built using the SpectrumSCM API, can apply business process decisions to each individual CR and assign them to the next responsible user, or place the CR in a holding pattern until certain business rules have been satisfied. For instance, a CR may not be eligible for promotion from the *Develop* phase into the *Test* phase until some form of code review has been performed. This diagram depicts the flow of control:



In this workflow, Change Requests are manually progressed out of each phase. When the Change Request enters the TBA super phase, the user defined automated workflow engine is immediately activated and decides where the Change Request should go next, and to whom it

[4] TBA = To Be Assigned

should be assigned. In this example, when a CR is progressed out of the ***Develop*** phase it is not immediately progressed into the ***Test*** phase. But rather it is left in the TBA super phase by the workflow engine until certain business rules have been followed.

14.4 API Activation Points

The SpectrumSCM API provides four separate activation points. The activation points can be used to enable a single automated extension, or can be used separately to implement independent extensions. Plugins that are associated with the activation points are run in separate threads of execution. Running the plugins in separate threads guarantees that the basic responsibilities of the SpectrumSCM server are never blocked.

14.4.1 System Startup and Shutdown

There are two activation points specifically designed to work with system startup and shutdown. When the SpectrumSCM server is started or stopped, all registered plugins are searched to see which ones implement the *SystemListener* interface. Each plugin that implements this interface is called from a separate thread of execution. It is possible that, given the length of execution of any particular startup or shutdown transaction, several plugin transactions may run concurrently.

- Example Usage

Use the startup and shutdown activation points to connect the SpectrumSCM server to external issue tracking systems or other external business systems.

14.4.2 Change Request Transition

Change Request transitions define the third plugin activation point. When a Change Request transitions from a user defined life-cycle phase into the TBA super phase, all plugins that implement the *ChangeRequestListener* interface are executed in separate threads. The change request transition activation point is one of the most useful activation points. Fully automated workflow systems will use this activation point as an event trigger for applying custom change request routing logic and other business processes.

- Example Usage

Use the change request transition activation point to implement a business rule sensitive automated workflow system.

14.4.3 Change Request Creation

Change Request creation defines the last plugin activation point. When a new Change Request is created in the SpectrumSCM system, this plugin activation point will be called in a separate thread of execution.

- Example Usage

Use the change request creation activation point to communicate change request creations to downstream project management systems.

14.5 Implementing the Interfaces

In order to create a user defined plugin, API users must first implement one or both of the Java interfaces defined by the API described below.

14.5.1 The *SystemListener* Interface

This interface defines the methods that are used by the startup and shutdown activation points. Users can implement this interface to create long running processes that are integrated directly into the SpectrumSCM server itself. For example, this interface could be implemented in order to create an active interface with an external system.
The following code block illustrates how to implement the *SystemListener* interface:

```
import scm.pub.interfaces.SystemListener;
public class WorkflowEngine implements SystemListener {
        public void startUp() {…}
        public void shutDown() {…}
}
```

14.6 The *ChangeRequestListener* Interface

This interface defines the methods that are called during Change Request creation and transition. The *ChangeRequestListener* interface defines the following two methods:

- changeRequestTransition(String project, ChangeRequest_d cr_d)
- changeRequestCreated(String project, ChangeRequest_d cr_d)

These two methods are passed the name of the project as a String and the data structure ChangeRequest_d. The ChangeRequest_d data structure contains all of the current information for the CR involved in the creation or transition. Users would need to write code similar to the following snippet in order to implement the ChangeRequestListener interface:

```
import scm.pub.interfaces.ChangeRequestListener;
import scm.pub.transport.ChangeRequest_d;

public class WorkflowEngine implements ChangeRequestListener {
        public void changeRequestTransition(String project, ChangeRequest_d cr_d) {
                …
        }

        public void changeRequestCreated(String project, ChangeRequest_d cr_d) {
                …
        }
}
```

14.7 Interacting with the System

There are three first class objects defined in the SpectrumSCM API that can be used to interact with a running SpectrumSCM server. These objects implement the *Proxy* design pattern[5] as described in the GOF (Gang of Four) design patterns book. Each object is a proxy or stand in for the corresponding persistent object located in the SpectrumSCM server.

- **ChangeRequest:** The ChangeRequest object is a proxy object for a live ChangeRequest object located in the SpectrumSCM server. Calling the getInfo( ) method on this object will result in all of the latest information for this particular Change Request to be returned. This object can be used to promote the Change Request into another phase or add history elements and other notes directly to the Change Request.

- **Project:** The Project object, just like the ChangeRequest object, is a proxy object for the live Project object located in the SpectrumSCM server. This object defines methods that allow the caller to extract project related information directly from the server. This object also contains methods that allow for the creation of new ChangeRequests.

- **ScmSystem:** The ScmSystem object, just like the other objects, is a proxy for the actual ScmSystem object. This object implements both the *Proxy* design pattern as well as the *Singleton* design pattern. The getInstance() method is used to retrieve **the one and only instance** of this object. The ScmSystem object contains methods that allow the caller to retrieve a list of all active projects in the system as well as a list of all registered system users. The object also contains an interface into the SpectrumSCM e-mail system, which allows the caller to send E-mail messages to interested parties.

A workflow engine can be designed to use these objects transiently for short term operations, or the objects, once constructed, can be stored at a higher scope level for use at a later time. The decisions for the design of the workflow engine are left up to the implementer.

14.8 Transport Objects

Transport objects are used as simple data structures to pass large amounts of information into and out of the proxy objects. The following is a list of all of the Transport Objects defined in the SpectrumSCM API.

- **AttributeMap_d:** An AttributeMap_d object is returned from the method Project.getProjectChangeRequestAttributes( ). The AttributeMap_d object is a mapping of Change Request attribute names to a set of attribute values.
- **ChangeRequest_d:** An ChangeRequest_d object is returned from the method ChangeRequest.getInfo( ). This data structure contains all of the current and historical information for the given Change Request.

---

[5] Gamma, Helm, Johnson, Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software. 1995 ISBN: 0-201-63361-2

- **ChangeRequestCreator_d:** This object is specifically used to create a new Change Request in the system. The contents of this object describe who the new Change Request will be assigned to, in what phase and on which Generic (branch).
- **ChangeRequestHistory_d:** The ChangeRequestHistory_d object is actually a sub-object that is returned as part of the ChangeRequest_d object. It contains historical information about the Change Request.
- **User_d:** An User_d object is returned from the method ScmSystem.getUserInfo( ) as well as Project.getUserInfo( ). In the case of calling the getUserInfo( ) method on the Project object, more information about the users current category assignments are returned.
- **CRFileDescriptor_d:** This data structure contains the version, generic and path information for a file under source code control. A set of CRFileDescriptor_d objects are returned from the method ChangeRequest.getFileDescriptors( )

14.9 Compiling the Code

In order to compile an API based plugin, the developer must have access to the SpectrumSCM server jar files. These jar files are located in the following directory:

<SCM_INSTALL_DIR>/lib

The developer's CLASSPATH environment variable must be extended to include the **scmServer.jar** file. The extended CLASSPATH variable should look like the following when complete:

CLASSPATH="$CLASSPATH:<SCM_INSTALL_DIR>/lib/scmServer.jar"
in Unix shell notation, or
CLASSPATH="%CLASSPATH%;<SCM_INSTALL_DIR>/lib/scmServer.jar"
in Windows batch notation.

Note that the path separators are platform dependent. Once the CLASSPATH variable is set properly, compile the code with the normal java compiler arguments.

14.10 Installing the Code

The developer's compiled code must be included in the SpectrumSCM server's CLASSPATH. A directory named *custom_plugins* exists in the SpectrumSCM server directory structure and default CLASSPATH, the developer's code can be placed in this directory.
<SCM_INSTALL_DIR>/SCM_VAR/custom_plugins

If the developer's code needs to reside in a jar file, the jar can be placed in the SpectrumSCM server lib directory. The script that is used to start the server must be modified to include this jar file. If the server is running on a Windows platform edit the file startServer.bat. If the server is running on a Unix or Mac platform, edit the file startServer.

14.11 Modify the Plugins XML file

In order to tell the SpectrumSCM server that a plugin has been added to the system, the plugins XML file must be modified. This file is located in the following directory:
<SCM_INSTALL_DIR>/SCM_VAR/etc/plugins.xml

The following is an example of a valid plugins.XML file:

```
<!-- This is an example of what the plugin file should look like -->
<!-- Note that the STATUS element can be set to either ENABLED OR DISABLED -->

<PLUGINS>
        <!-- Put your plugin declarations here -->
        <PLUGIN>
                <NAME VALUE="Coreys Plugin"/>
                <CLASS VALUE="com.scm.TestPlugin"/>
                <PROJECT VALUE="SCM"/>
                <STATUS VALUE="ENABLED"/>
        </PLUGIN>
</PLUGINS>
```

There are four (4) XML elements that must be in this file. Each element is described below:

- **NAME:** This is simply the name of the plugin and is used initially as a key to the plugin itself.
- **CLASS:** This is the actual class name of the class that implements the listeners described above.
- **PROJECT:** This is the Project name that this particular plugin should be associated with. The same plugin can be associated with separate Projects as long as the developer is careful not to include Java class attributes with global class scope in the plugin.
- **STATUS:** This determines whether the plugin should be used or not. Set this to DISABLED if the plugin needs to be turned off.

14.12 Example plugins

The basic plugin skeleton is trivial to construct. Here is a working plugin that implements both interfaces but doesn't really do anything:

```
import java.io.*;

import scm.pub.interfaces.*;
import scm.pub.transport.*;
import scm.pub.exceptions.*;

public class TestPlugin implements SystemListener, ChangeRequestListener {

        public void startUp() {
                System.out.println("Startup called..");
        }

        public void shutDown() {
                System.out.println("ShutDown called..");
        }

...
```

```
…
        public void
        changeRequestCreated(String project, ChangeRequest_d cr_d) {
                System.out.println("changeRequestCreated..");
        }

        public void
        changeRequestTransition(String project, ChangeRequest_d cr_d) {
                System.out.println("ChangeRequestTransition called...");
        }
}
```

Note that the only thing this plugin does is report to the standard output when the interface methods have fired. Compile and add this plugin to the system to see which user level actions cause these methods to execute. For instance, creating a new Change Request will cause the changeRequestCreated() method to execute. Progressing that CR into the TBA state will cause the changeRequestTransition() method to execute. The start() and stop() methods will execute when the server is started and stopped.

This next code snippet accesses all of the major first class objects and extracts some information from a Change Request:

```
        public void
        changeRequestCreated(String project, ChangeRequest_d cr_d) {
                java.lang.System.out.println(cr_d.toString());

                try {
                        ScmSystem       sys     = ScmSystem.getInstance();
                        Project         proj    = new Project(project);
                        ChangeRequest   cr      = new ChangeRequest(proj, cr_d.getCRId());

                        System.out.println(cr.getInfo().toString());

                } catch(Exception e) {
                        System.err.println("Caught: " + e.getMessage());
                }
        }
```

Note that this code is actually redundant. The ChangeRequest_d information that was extracted from the cr.getInfo() call was already handed to the enclosing method as an argument. The example is just to show how to access some of the more important objects. Also notice that the System object is accessed by simply calling the static method getInstance() on the System class. Projects and ChangeRequests can be constructed as often as necessary. All of these objects can be stored for later use once they have been constructed.

This next example is a more complete example of an automated workflow engine. In this code snippet, the ChangeRequest passed to the transition method is examined and automatically progressed into the next life-cycle phase:

```
public void changeRequestTransition(String project, ChangeRequest_d cr_d) {

        ScmSystem       sys    = null;
        Project         proj   = null;
        ChangeRequest cr       = null;
        Vector       phases    = null;
        try {
                sys    = ScmSystem.getInstance();
                proj   = new Project(project);
                phases = proj.getLifeCyclePhases();
                cr  = new ChangeRequest(proj, cr_d.getCRId());
        } catch(Exception e) {
                System.err.println("Caught: " + e.getMessage());
                return;
        }

        try {
                ChangeRequest_d          crObj = cr.getInfo();
                ChangeRequestHistory_d crh_d = null;

                Vector        history    = crObj.getHistoryInfo();
                String        lastPhase  = null;
                String        nextPhase = null;
                int           index      = -1;

                for(int indx = history.size() - 1; indx >= 0; indx--) {
                        crh_d = (ChangeRequestHistory_d)history.get(indx);
                        if(crh_d.getPhase().endsWith("note")) {
                                continue;
                        } else {
                                lastPhase = crh_d.getPhase();
                                break;
                        }
                }
                index = phases.indexOf(lastPhase);
                nextPhase = (String)phases.get(index+1);
                cr.assignToPhase(crh_d.getUser(), crObj.getCurrentGeneric(),
                                 nextPhase, "Here's some more work");
                sys.sendEMail("joe@x.com", "CR Status", "Assigned CR <" +
                                        crObj.getCRId() + "> to phase <" +
                                        nextPhase + ">");
        } catch(Exception e) {
                System.err.println("Caught: " + e.getMessage());
        }
}
```

Unfortunately, in order to get all of the code into this single example, some of the empty lines had to be removed from the text and the vast majority of the error handling code has also been removed. The last few lines in the example are the most important. The method assignToPhase() called against the ChangeRequest actually assigns this particular CR to the next phase in the life-cycle and adds a small note. The next line uses the e-mail interface to send mail to an interested party.

14.13 Summary

The SpectrumSCM API allows a developer to easily create fully automated business processes and external system interfaces. Currently the API is limited to this type of functionality. The developers of the API chose to exclude a file level listener interface from the current API implementation. The existence of such a listener has limited use in a fully integrated tool like SpectrumSCM. One of the basic tenets of SpectrumSCM is that individual files are worked or changed as part of a larger issue or change request. In this scenario, the need to know when a single file has changed, or to act upon a single file change is unnecessary. In other systems that are *interfaced* instead of *integrated,* this type of functionality may be necessary as individual file changes are not already associated with a traceable statement of work. File information can be retrieved through the corresponding Change Request.