# Sample Tracking

## JIRA Sub-Task Plug-in

# The Existing System

## Problems with the existing method of Sub-Task Creation

### A: Inconsistent sub-task creation

Sub-Tasks can be created in two different ways, add Sub-Task and create Sub-Task.

The 'add Sub-Task' method uses a form embedded in the Issue View page. It is only available once an Issue has at least one sub-task.

The fields in this form ignore the settings of the Create Issue/Sub-Task screen and instead the same set of fields are used for all projects in the system. The created Sub-Tasks do not necessarily include all of the important/required fields. Field validation added to the 'Create' transition of the workflow appears to be ignored.

### B: No synchronization of sub-task creation with the Workflow

The 'Sample Tracking' system only uses sub-tasks during two of the workflow steps. To avoid confusion it would be best if the sub-tasks that can be added to an Issue were dependent on the workflow step that the Issue was on.

The workflow is the natural place to put logic that is based on the step the Issue is on.

### C: All sub-tasks are listed, for creation, all the time

The list of sub-tasks used by both the 'create sub-task' and 'add sub-task' pages contains all of the sub-tasks mapped to the project and all of the sub-tasks that were globally created.

The sub-tasks used/available are specific to the issue's current workflow step. Ideally only the sub-tasks that should be used at that step should be listed.

The sample tracking system is intended to be used by a number of groups (sequencing, PI, closure, annotation...). Ideally the impact of changes should be limited to the fewest people possible. By separating the sub-tasks available at each step the addition and removal of new steps will only effect those who use that step.

# Analysis of how Sub-tasks are created in JIRA

## 'Create' Sub-task

### Web Form

Opens a separate page with a drop-down list of the available Sub-Tasks.

The form submits to */secure/CreateSubTaskIssue.jspa.* It, and the form for 'add' sub-task, contain the following elements:

- Select tag with name = 'issuetype'

  The options are the sub-tasks, defined with

  - **value** = *IssueConstant.getID()*
    Format: '*[0-9]{2}*'
    Example: **36**
  - **style** = **'background-image: url(***IssueConstant.getURL()***)'**
    Format: */[a-zA-Z0-9]+)+[.]png*
    Example: **/images/icons/book.png**
  - content = *IssueConstant.getName()*
    Example: **Library Construction Task**

- Hidden tag with name = '**parentIssueId**' and
  Value: *Issue.getID()*
  Format: *[0-9]{5}*
  Example: **22986**

- Hidden tag with name = **'pid'**
  Value: '*project.getID()*'
  Format: [0-9]{5}
  Example: **10180**

The following elements are only in the 'Create' Sub-task form:

- Submit tag with name = 'Next>>'
  Value: 'Next>>'

After a Sub-Task type has been selected the user proceeds to a form containing the fields defined in the screen linked to the Sub-Task in the Issue Type Screen Scheme.

## Code path

***Note: This is only for Jira 4.0 and earlier. Jira 4.1+ use Web-Sections and Web-Items.***

### Velocity Template: IssueSummary.vm

The side bar is generated by *IssueSummary.vm*. The link for 'Create subtask' is in the operations section. The operations section is wrapped with "*if '$layout.showIssueOperations*"'. The test for this is implemented in *IssueSummaryLayoutBean.isShowIssueOperators.*

The contents of the operations section is generated by iterating through the values in the context variable "*$issueOperations*". The contents of the variable is generated by *IssueSummaryWebComponent.getViewableOperationDescriptors(Issue)*. The method requests from the plug-in registry all instances of *PluggableIssueOperation*. A plug-in is only added to the set if *showOperation(Issue)* returns *true*.

A plug-in for 'create subtask' is never directly requested, it has already been registered with the plug-in manager. The definition used to register the plug-in comes from:

*WEB-INF/classes/system-issueoperations-plugin.xml*

### Issue-operation Plugin: 'Create Subtask'

The plug-ins definition in *system-issueoperations-plugin.xml is:*

```
<issue-operation
             key            ="create-subtask"
             i18n-name-key  ="admin.issue.operations.plugin.create.subtask.name"
             name           ="Create subtask for this issue"
             class          ="com.atlassian.jira.issue.operations.CreateSubtaskOperation"
             state          ='enabled'>
        <resource
                  type       ="velocity"
                  name       ="view"
                  location   ="templates/plugins/operations/basic-operation.vm" />
       <order>60</order>
</issue-operation>
```

The XML file gives both the action class and the view template. The view template ***basic-operation.vm*** contains:

```
$i18n.getText(
      $i18nDescKey,
      "<strong> "+
            "<a id='$aId'
              href='$requestContext.baseUrl$actionUrl'
               rel='lazyLink'>",
            "</a> +
      "</strong>")
```

The template produces a link wrapped in '*strong*' tags. The contents of the link is *$i18nDescKey* which is looked up using *$i18n.getText*. The target of the link is *'$requestContext.baseUrl$actionUrl'*. The *$i18n.getText* and *$requestContext.baseUrl* are part of the standard velocity environment. *$18nDescKey* and *$actionUrl* are specific to the implementation of the operation.

### Method call: CreateSubtaskOperation.showOperation

The plug-in contains a test to decide if it should be included in the page. The tests carried out in *showOperation* are identical to those of *AbstractViewIssue.isSubTaskCreateable* except that when the project permission, CREATE_ISSUE, is tested it is done so without a *User* object. This appears to be because of limitations of the interface, showOperation is only passed an issue and not a user object.

## Add Sub-task

### Web Form

This option only becomes available once a Sub-Task has been added to the Issue. It is part of the section displaying the Issues sub-tasks. It includes a drop-down to select the Sub-Task without going to a separate page. It also shows a limited set of fields from the Sub-Task. This form can be customized by providing a list of fields in the jira.subtask.quickcreateform.fields property of the jira-config.properties file. The value is shared by all projects and so it is unlikely to be an acceptable solution.

The add Sub-Type form submits to */secure/CreateSubTaskIssueDetails.jspa.* It shares '**issuetype**','**parentIssueId**' and '**pid**' with the 'create' sub-task form. It adds the following extra fields:

Input tag with name = 'summary'

- Select tag with name = **'assignee'**
  - Option tag with value = "" for unassigned
  - Option tag with value = -1 for auto
  - Option tags with values = *usernames* ...
- Input name = **'timetracking'**
- Hidden tag with name = **'viewIssueKey'**

  Value:

  Format: '[A-Z]{2,}-[0-9]{3}

  Example: **ST-605**
- Hidden tag with name = **'quickCreate'**

  Value: **'true'**
- Submit tag with id = 'stqc_submit'

  Value = ' Add '

### Code path for 'add subtask'

#### JSP: view_issue

The sub-task display within *view_issue.jsp* is created by: *includes/panels/issue/view_subtaskissues.jsp*

#### JSP: view_subtaskissues

The add subtask button is wrapped with a webwork's if tag.

```
<webwork:if test="/subTaskCreatable[I 1.0] == true">
     <webwork:property value="/subTaskQuickCreationWebComponent/html()" escape="'false'" />
 </webwork:if>
```

The same if tag is used in several other places, such as at the start and finish of the web-form

I 1.0 Modifying isSubTaskCreatable would allow the form to be controlled.

## Method call: AbstractViewIssue.isSubTaskCreateable (from View Issue)

*"/subTaskCreateable"* is resolved by looking for a method called *isSubTaskCreateable()* on the action object. For *view_issue.jsp* the action object is *com.atlassian.jira.web.action.issue.ViewIssue*[I 1.1]. The method '*subTaskCreateable*'[I 1.2] is actually implemented several levels back in the hierarchy in *com.atlassian.jira.web.action.issue.AbstractViewIssue*. It is inherited through:

- *com.atlassian.jira.web.action.issue.AbstractViewIssue*
- *com.atlassian.jira.web.action.issue.AbstractCommentableIssue*
- *com.atlassian.jira.web.action.issue.AddComment*
- *com.atlassian.jira.web.action.issue.ViewIssue*

The implementation carries out the following checks:

- Tests that sub-tasks are enabled

  By checking *APKeys.JIRA_OPTION_ALLOWSUBTASKS* (***jira.option.allowsubtasks***) is set. It looks first into its database backed properties store and if nothing is found tries loading it from *jira-application.properties*.

  Not practical to intercept, values are only read from the properties file on start-up and then only if the property isn't already in the Jira configuration database. It is also shared across all projects and issues.

- Tests that this issue isn't a sub-task

  By querying the Sub-Task Manger, with the current issue. The sub-task manager (*com.atlassian.jira.config.DefaultSubTaskManager*) tests if the issue has a parent. It does this by finding any links that have the issue's id as their destination and then testing each to see if it is a sub-task link. *DefaultIssueLinkManager* carries out the actual locating of the links.

By adding a link of type sub-type to the issue it will stop sub-task creation from being available. Removing the link would return the ability[I 2.0]. It is issue dependent and the addition or removal of the link could be a workflow function.

- Tests that the current user has permission to edit the issue

  This is actually two tests. First the user's permissions for the issue, set via the Permission Scheme, are checked. The permission tested is *Permissions.EDIT_ISSUE* (12). *AbstractPermissionManager* is passed the permission, the issue and the user; via *AbstractIssueSelectAction*. The *PermissionsManager* tests that the user has the permission within the project and then tests with *IssueSecuritySchemeManager-IssueSecuritySchemeManager* that the Issue does not have a security restriction which bars the user from using that issue.

  If the user has the required permission then the workflow state is checked to determine if the permission has been revoked. The query is passed, via AbstractIssueSelectAction to DefaultIssueManager, to SimpleWorkflowManager. The issue's workflow is accessed via the project id and the issue id. **The current step is determined using the issues status (Non-unique statuses will cause problems with this).** The meta attributes of the step are tested to see if *JiraWorkflow.JIRA_META_ATTRIBUTE_EDIT_ALLOWED* (jira.issue.editable) is equal to

I 1.1 Overriding the action of ViewIssue to use a different action class (one that sub-typed ViewIssue)
I 1.2 Modifying a property or attribute to cause the existing isSubTaskCreateable test to fail.
I 2.0 Modifying the Issue to appear to be a sub-task

'false'. Unlike other workflow permissions no user or user group can be used with this permission.

Modifying the Edit Issue permission can be done on an issue by issue basis dependent on workflow state, but if the user can't edit the issue its not much use.

- Tests if the user has the project permission CREATE_ISSUE for this project

  The project permission Permissions.CREATE_ISSUE (11) is tested for by passing it and the project to AbstractPermissionManager via JiraWebActionSupport. The Permission Scheme is used to determine if the user has the permission.

  This permission is at the project level and so not specific enough for our needs.

- Tests if the issue type field is set[I 2.1]

  I don't know what it should be set to for a normal issue. It may be possible to copy/move the options to another field. Without knowing what they are or what they are used for deleting them could be foolish.

## SubTaskQuickCreationWebComponent

The Html is generated by com.atlassian.jira.web.util.SubTaskQuickCreationWebComponent.

```
<webwork:property value="/subTaskQuickCreationWebComponent/html()" escape="'false'" />
```

The properties value becomes a call to *getHtml* on *SubTaskQuickCreationWebComponent*. The method sets up the normal JiraVelocity parameters and also places "*displayFieldIds*" and "*presetFieldIds*" in the velocity environment. It then delegates to the velocity template. The template used is determined by "*jira.subtask.quickcreateform.template*[I 3.0]"

The value for the template is defined in the configuration for the whole system. There is no way to assign values on a project by project basis. The intercept would need to check the project itself and then load the old template for other projects.

Values for the velocity template are:

*templates/jira/issue/subtask/quickcreationform-vertical.vm*

*templates/jira/issue/subtask/quickcreationform-horizontal.vm*

The input button is hard coded in them. The vertical version does contain javascript to hide and reveal the form though.

One of the form elements, in both versions, is:

```
<input type="hidden" name="quickCreate" value="true" />
```
[I 3.1]

The templates make multiple call backs to *SubTaskQuickCreationWebComponent*:

- *$webComponent.getSubTaskFieldHtml*
- *$webComponent.getParentIssue*
- *$webComponent.getSubTask*
- *$webComponent.getSubTaskFieldPreset*

None of them, however, effect the 'add subtask' button.

---

I 2.1 Making the issue type field null or empty.
I 3.0 Modify the parameter to change the Velocity Template used
I 3.1 Not actually an intercept. This is used by the page the form submits to, not by the page the form is on.

## Shared

### Class: CreateSubTaskIssueDetails

This is the action used when sub-tasks are normally created. It may be possible to reuse some or all of this to carryout the actual sub-task creation.

The bean methods are (bold indicates added in this class, instead of inherited):

- setAssignee(<String>)
- setCommand(<String>)
- setId(<Long>)
- **setIssuetype(<String>)**
- setKey(<String>)
- **setParentIssueId(<Long>)**
- setPid(<Long>)
- **setQuickCreate(<boolean>)**
- setReturnUrl(<String>)
- setSelectedIssueId(<String>)
- setSelectedProjectId(<Long>)
- setViewIssueKey(<String>)

The other setters are:

- setCurrentIssue(<GenericValue>)
- setErrorMessages(<Collection>)
- setErrors(<Map>)
- setHistoryIssuetype()
- **setIssue(<GenericValue>)**
- setSearchRequest(<SearchRequest>)
- **setSelectedProject(<GenericValue>)**
- **setSelectedProject(<Project>)**

Other fields are needed though depending on what sub-task is being created. See the creation process to see if the form can be reused in the bulk version. The form is CreateSubTaskIssue.jspa and submits to CreateSubTaskIssueDetails.jspa

### JSP: secure/views/createsubtaskissue-details.jsp

Creates the form where the parameters are set. Most of the actual form seems to be generated by /templates/standard/issuefields.jsp. It takes the following parameters:

- issue
- tabs
- errortabs
- selectedtabs
- ignoredfields
- create

In the JSP they are set using the webworks Param tag within the webworks Include tag. The underlying implementation simply appends the parameters to the URL before passing it to the ServletRequest to get a 'RequestDispatcher'.

# Possible Solutions

## Using Permissions

The 'create subtask' and 'add subtask' elements both check if the user has permission to create an issue before adding themselves. In a permission scheme it is possible to allocate a permission based on the value of a custom form element.

Jira allows this to be set, but then appears to ignore it.

Also tried creating a normal user (username=nonadmin password=password) to see if it was administrator privileges overriding the setting. The results were the same though.

(also tried disabling via workflow properties)

## Handling form generation

1. Modify the form to redirect to the same place as create sub-task [Solves A]
2. Stop the 'add subtask' button being created (Server side) [Solves A]
3. Stopping the 'add subtask' button being displayed (Client side) [Solves A]
4. Modify the issue to disable all sub-task creation (fake being an existing sub-task) [Solves A]
5. Use the workflow to fill in any missing fields, that can be auto filled [Solves A]. Limited to situations where the custom fields are all automatically set.
6. Use a workflow state to show that the Sub-Task still needs further setup [Solves A]
7. Adding a 'security' property for create sub-task and then using the workflow to revoke it.

## Turning the issue into a pseudo sub-task

The code paths for 'add subtask' and 'create subtask' provide two conditions that are checked:

- Incoming links of type sub-task
- Null or empty value for the issue-type field

Adding a link seems like the safer option.

*CreateSubTaskIssueDetails* creates the sub-tasks for 'create subtask'. It does this by first creating an issue, as normal and then linking the created issue to its parent. The linking is done by

*(Default)SubTaskManager.createSubTaskIssueLink* with parameters:

- the parent issue
- newly created sub-task
- the user

The sub-task manager calls *(Default)IssueLinkManager.createIssueLink* with:

- parent Issue's ID [*parentIssue.getLong("id")*]
- the sub-task's ID [*subTask.getLong("id")*]

- the link type [*IssueLinkManager.getIssueLinkTypesByStyle(SUB_TASK_LINK_TYPE_STYLE)*]
- the sequence, or counter, from the parent issue for numbering its links
- the user

The actual creation doesn't test if the parent is a sub-task itself. Causing the display add/create sub-task tests to fail should not inhibit actually creating sub-tasks.

Note: SubTaskQuickCreationConfigImpl also contains an implementation of the field loading code in parseFieldIds. It currently doesn't reference the Issue Type or the Project.

## Sub-Task creation via Workflow actions.

The workflow is the natural place to put logic that is based on the step the Issue is on. By controlling the creation of the Sub-Tasks it would be possible to have a single action to create a Sanger Sequencing task.

It is possible to create a Sub-Task via the 'Create Sub-Task on Transition' workflow function plug-in.

http://confluence.fangwai.net/display/JIRAEXT/Create+Sub-Task+on+transition

The problems are

- The sub-task is created only using the existing information
- The existing create Sub-Task links will need to be hidden
- An action is needed for each type of sub-task.

The first problem, only using existing information, requires a method of presenting the user with a form to fill in the missing information on the issue.

The following were attempted:

- Adding a form to the creating transition only edits the parent issue.
- Adding a form to the Sub-Tasks creation is ignored.
- Adding an automatic transition after creation and adding a screen to that results in the transition taking place but the screen is never displayed.

### Using a validator

Validators are passed the contents of the transition form. If it is possible to add extra fields to this screen, without altering the existing issue, then the transition screen could be modified to become the create sub-task screen.

The extra fields could be added by a custom field that generated them based on the fields associated with the sub-task. To do this however it would need to know what the sub-task was.

- Is it possible to tell which transition the screen is being view for?
- Is it possible to tell what validators will be used when generating the form?

  (this could be used to query the validator's configuration to find out which sub-task to present).

## Using workflow permissions for limiting sub-task creation

- Create Issue: project permissions don't check the workflow.
- edit Issue: this can be set to false, but the no-one can change the issue.

## Using a custom field to control permissions

In the permission scheme it is possible to specify a custom field to use as the look-up for which users have permission X. This allows issue specific permissions. It is not clear how much information would be passed to the custom field in order for it to determine who should have a permission. The permission scheme ideally would be defined in the workflow and the field would somehow interface with it to determine what to do. (See workflow permissions and how the workflow is accessed).

Unfortunately while the permission scheme can be set to have Create Issue depend on a custom field all that happens is that everyone is given the create permission.

## Disabling sub-task buttons via Javascript

Javascript can be added to the view issue page via Web Item Plugin Modules[1]

*http://confluence.atlassian.com/display/JIRA/Web+Item+Plugin+Module*

A custom field could be used, with its view template containing the Javascript. It would also need an edit/create template that set a dummy value to ensure it was added to the view issue page.

## Using a custom field and a transition screen

By having the custom field insert a new form that acts like the select sub-task page of the create sub-task path we could bypass the problems of the validator based solution.

- No need to know what transition is being performed
- No need to tell which fields to display for the sub-task
- Fewer extra transitions.

The sub-task creation custom field would be configured with the list of sub-tasks that it should show. It would then be hidden on all but one transition screen. A transition using that screen would then become the 'create sub-task' button. Multiple sets of sub-tasks could be created as separate custom field instances, with separate configurations.

A method of disabling the normal sub-task buttons on the issue view would be needed.

The custom-field would need to hide the submit button which appears to be given the same name as the transition and so confusingly would state 'Create Subtasks'

This could be used to our benefit, if the custom field added something like

```
</form>
<form action="createSubTaskPage.jspa">
      <input type="hidden" name=...
```

1 pre 4.1used http://confluence.atlassian.com/display/JIRA/Issue+Operations+Plugin+Module

```
<select ...
      <option ....>Sub-task type 1</option>
      ....
</select>
```

Then the buttons would be part of the new form and submit to the createSubTaskPage.

The downside is that the available sub-tasks would not be defined in the workflow. They would be defined in a combination of the Custom field's configuration, the screen's settings and the transition in the workflow.

Each new group of sub-tasks would require a new instance of the custom field and a new screen defining.

## Using Javascript to redirect the creation buttons

This allows a new action to be used instead of intercepting the current one. The advantage of this is that if it should fail the user will be able to continue working, all be it having to remember not to click 'add subtask' and selecting from a longer list of sub-tasks.

See the search doc for details on the Javascript

## Project / Step Specific

The current Issue ID is passed to the Sub-Task selection servlet, which is in turn passed on to the Sub-Task creation servlet. The state can be used to look up the step.

Alternatively a hidden Custom-Field could be used to communicate between the workflow and the sub-task via the Javascript that modifies the URL.

# Bulk sub-task creation

The core interface for Bulk Operation Plug-ins is Bulk Operation. There isn't a plug-in descriptor for Bulk Operations and so the class has to be manually registered with the Bulk Operation Manager. (This requires a static constructor in another class that the plug-in framework will automatically load). Registering the Bulk Operation class adds it to the list of options presented to the user. If the operation is selected the user is redirected to "getOpertationName()"+details.jspa . An action in a webworks actions section that matches the URL up to the .jspa is required. The action should extend AbstractBulkOperationDetailsAction. This class provides utility methods for accessing the BulkOperationBean which holds the active state of the action. The doDetails method should return a view that provides any plug-in specific information. There are no restrictions of what commands are provided and what they are used for but eventually the perform method on the Bulk Operation class should be called to carry out the work.

## Analysis of Bulk Operations within JIRA (Code)

### BulkOperationManager

BulkOperation objects register with the BulkOperationManager (implemented by DefaultBulkOperationManager).

There seem to have been some been problems with attempting to call addBulkOperation and so some have implemented their own version, that includes the extra operation(s).

### BulkOperation

This is the 'descriptor' class. It gets registered with the BulkOperationManager and is used to start the whole bulk update process. The 'canPerform' method is used to decide if the operation should be available. If 'canPerform' returns false then 'getCannotPerformMessageKey' is used to inform the user why the operation is unavailable.

The 'perform' method is somewhat out of place among the view methods. It actually carries out the operation. Strangely it appears to always be called from the Action (doPerform) which is part of the plug-in. There was no need to specify it as part of this interface.

#### Presentation Methods

The interface only provides customization of the text shown, no templates it appears.

- String getCannotPerformMessageKey()
- String getDescriptionKey()
- String getNameKey()
- String getOperationName()
  I think this may be used to create the URL that the request is submitted to

Note: in version 4.3+ User is replaced by Userremote

### Actions

#### boolean canPerform(BulkEditBean, User)

Determines whether the operation can be performed with the given set of issues

#### perform(BulkEditBean, User)

Performs the operation on the given set of issues. BulkEditBean.getSelectedIssues is used to get a List of issues to loop over. The objects in the list are Issue objects. As the operation could be running for a long time the issue should be tested again before it is acted on. Bulk delete uses the Issue Manager, if the Issue Manager returns null to a request for that Issue's ID then it no-longer exists and should be skipped.

Bulk Delete uses a somewhat complex call to carryout the delete:

```
Map parameters =
      EasyMap.build("issue",               issue.getGenericValue(),
                    "remoteUser",          remoteUser,
                    IssueEvent.SEND_MAIL,  Boolean.valueOf(sendMail))
ActionDispatcher dispatch = CoreFactory.getActionDispatcher();
ActionResult aResult = dispatch.execute(ActionNames.ISSUE_DELETE, parameters);
ActionUtils.checkForErrors(aResult);
```

ActionDispatcher appears to be a method of 'calling' an action as though a request was submitted to it. (struts has a class called ActionMapping to do this. It requires an ActionForm, HttpServletRequest and a HttpServletResponse object as parameters though). Internally a GenericDispatcher is created (using the first parameter, the action's name). The dispatcher is prepared by attaching the user and parameter values passed. The action is done via calling executeAction and the ActionResult is passed back from the 'finish' method.

The Action Dispatcher may be in package com.atlassian.webwork1.action for older versions.

There doesn't appear to be a simple way to tell what parameters are needed by an action.

## BulkEditBean

The BulkEditBean appears to be important to the operation, the operation must access all of its parameters (other than user) via it.

## AbstractBulkOperationDetailsAction

These appear to be Struts type Actions, which combined with templates provide the interface to the bulk operation. The class hierarchy is quite deep

- `jira.web.action.issue.bulkedit.AbstractBulkOperationDetailsAction`
  Adds three, un-implemented, actions (details,detailsValidation and perform)
  Its constructor takes a SearchService possibly for selecting the issues?
- `jira.web.action.issue.bulkedit.AbstractBulkOperationAction`
  Provides methods to get and clear the BulkEditBean
- `jira.web.action.IssueActionSupport`
- `jira.web.action.ProjectActionSupport`
- `jira.web.action.JiraWebActionSupport`
- `jira.action.JiraActionSupport`
- `webwork.action.ActionSupport`

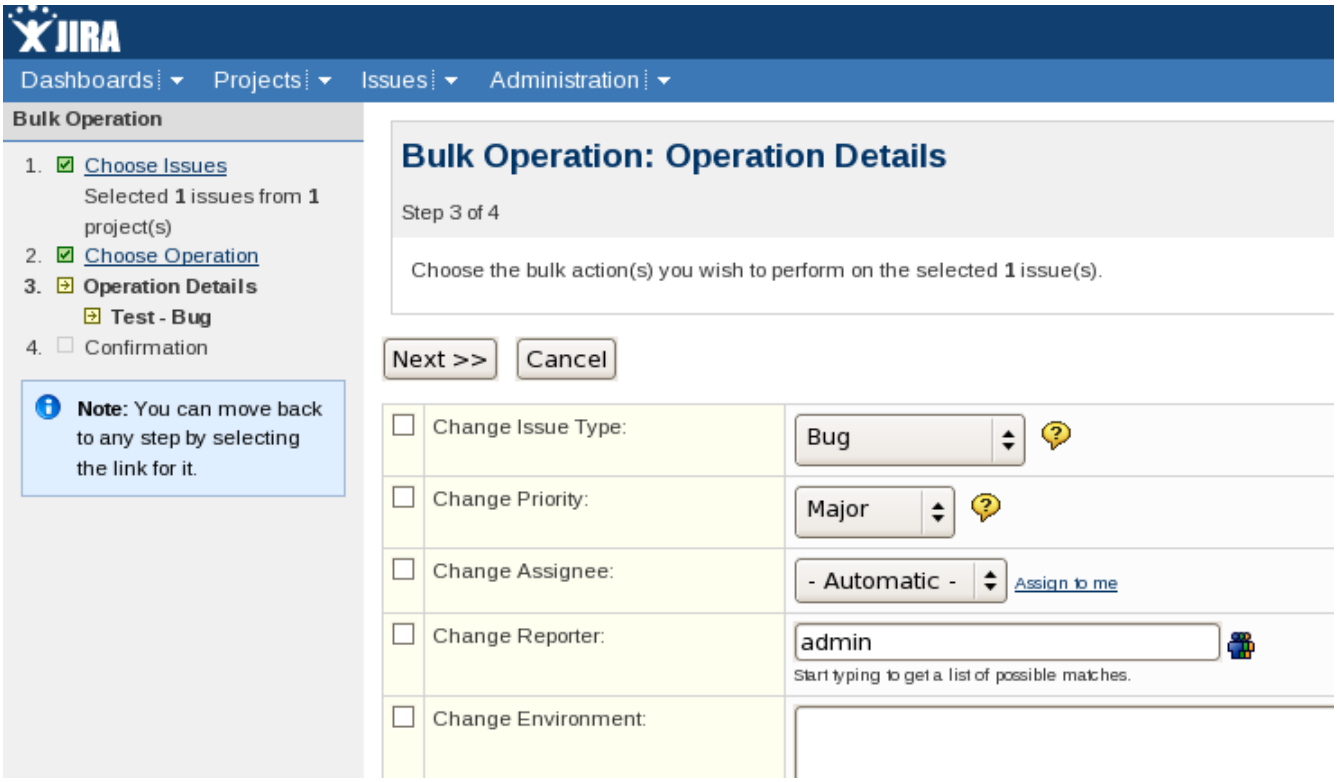
Only the last two classes are specific to Bulk operations though.

The actions and views appear to be similar to configuring a custom field, in that there isn't much of a framework there. The plug-in can do whatever it likes from the point where it is selected as the operation to use onwards.

## Analysis of Bulk Operations within JIRA (HTML)

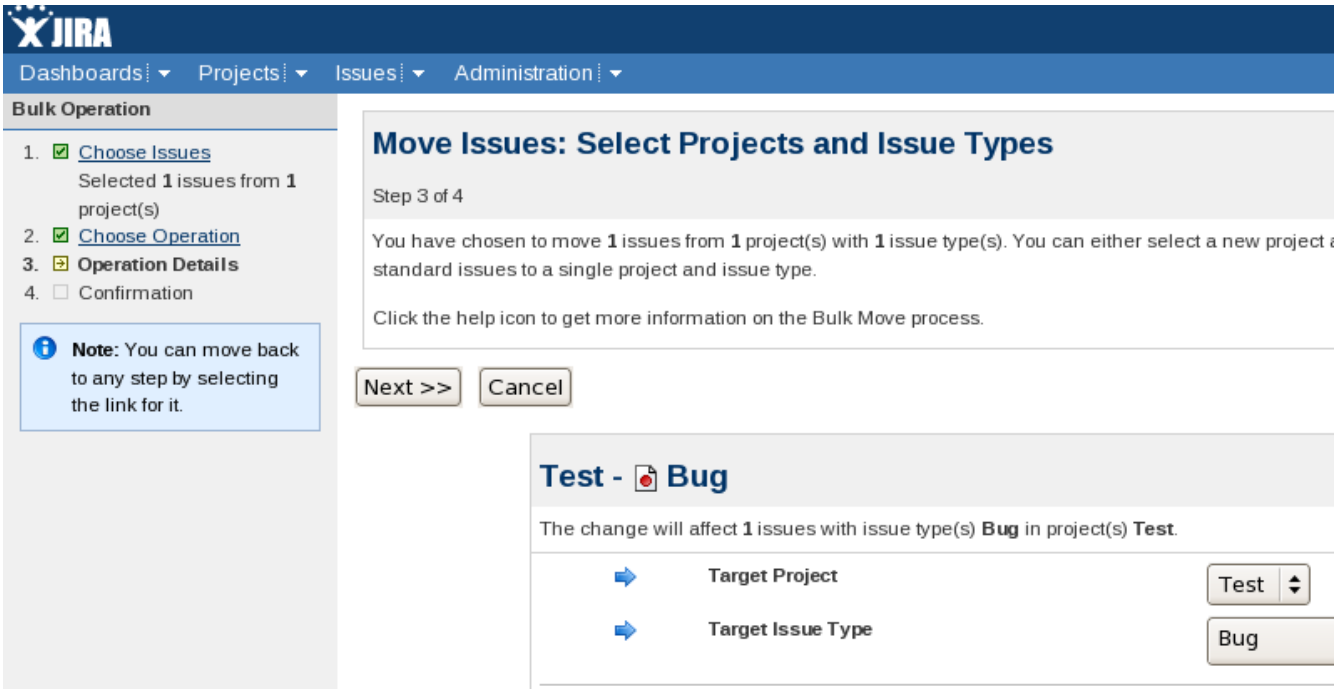
The Page layout for the bulk operations isn't very standardized:

### Bulk Edit



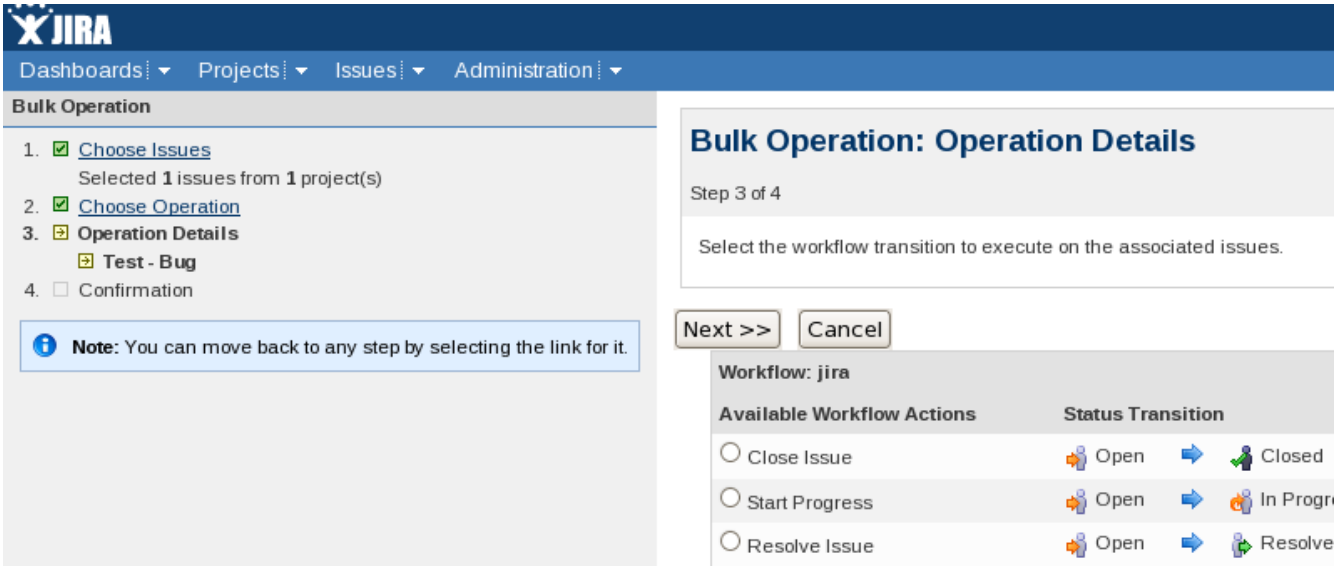

- The area with the fields is wider than the top title box.
- The heading doesn't indicate the operation that was chosen
- The current step shown to the left is a sub-step of 3.

## Bulk Move




- The middle box is much narrower than the top one
- There is a second title in the middle box
- The current step is 3.

## Bulk Transition




- The left panel has grown, breaking the layout
- The heading doesn't indicate the operation that was chosen
- The middle panel is slightly smaller than the top
- The current step shown to the left is a sub-step of 3.

### Bulk Delete

Doesn't appear to have an Operation Details screen and jumps to confirmation

### Summary

The numbering of the steps isn't very consistent, the layout varies significantly. The common parts are:

- The left bar (despite changing shape and numbering)
- The title box (although the information in the title changes between implementations)
- The next and cancel buttons

## The left bar

This is generated by /secure/views/bulkedit/bulkedit_leftpane.jsp

The format is quite restrictive, the number of steps and the names of the steps are fixed. The main interaction with the form is determining which is the current step and which steps have been completed. This is done via web work tags:

```
<webwork:if test="/rootBulkEditBean/currentStep == 1">
<webwork:if test="/rootBulkEditBean/availablePreviousStep(1) == true">
```

The sub-steps that appear under step 3 in some of the operations are a list of Project, Issue Type pairs for operations like Edit and Transition where the options they display are dependent on both the project and the issue type. They create a series of sub-groups, one for each unique pairing of issue type and project in the selected issues.

The menu is displayed if:

```
<webwork:if test="!@hideSubMenu">
```

It loops over the sub-types using:

```
<webwork:iterator value="./bulkEditBeans" status="'status'">
```

The class of the entry, which is used with the style sheet style to render the html, is set by one of:

```
<webwork:if test="@status/index < ../currentBulkEditBeanIndex">  class="done"
<webwork:if test="@status/index == ../currentBulkEditBeanIndex"> class="current"
<webwork:else> class="todo"
```

The actual entires are simply <project> - <IssueType>

```
<webwork:property value="./key/project/string('name')" /> -
<webwork:property value="./key/issueTypeObject/name" />
```

The links back to previous sections are also mostly fixed:

1. `BulkEdit1!default.jsp?currentStep=1 (if maxIssues > 0)&tempMax=<webwork:property value="/rootBulkEditBean/maxIssues"/>`
2. `BulkChoose!default.jspa`
3. `<webwork:property value="/operationDetailsActionName"/>`
4. `Never has a link`

3 Is the the only one that can be modified.

### Requirements

- <BulkEditBean> getRootBulkEditBean()
- <String> getOperationDetailsActionName()

All of the other calls are to the BulkEditBean

## Next and Cancel Buttons

The buttons are in the body of the form which is hard to copy if the form is generated using existing JSP pages.

```
    <input type=submit id="Next" name="Next" value="Next >>" accessKey="S" title="Press
Ctrl+Shift+S to submit form">
     <input type=button id="Cancel" name="Cancel" value="Cancel"
onclick="location.href='BulkCancelWizard.jspa'">
```

The cancel button doesn't appear to be using the form and so will be easy to move.

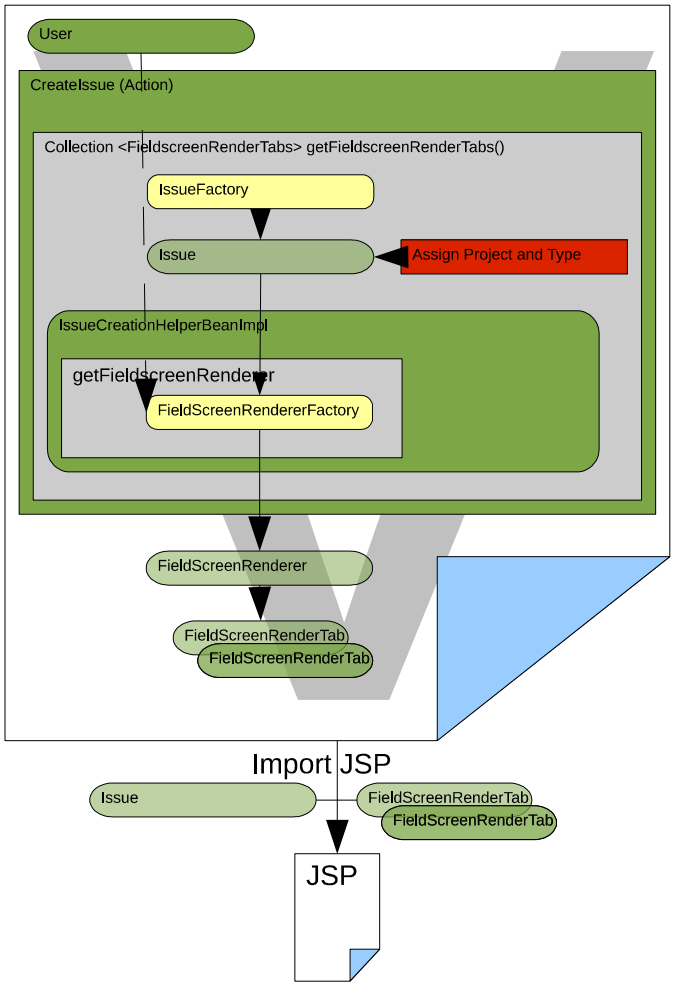
# Issue creation display

The fields displayed are determined by the elements of the tabs collection passed to the issuefields jsp. The elements are of type FieldScreenTabRenderTab.

The CreateIssue Action has a method 'getFieldscreenRenderTabs' that is used to populate the tabs parameter. CreateSubtypeIssueDetails delegates to this method to get the tabs and input fields for the subtask. It gets its values from calling 'getFieldscreenRenderTabs' on the FieldScreenRenderer, accessed via 'getFieldscreenRenderer' on the CreateIssue Action.

The FieldscreenRenderer comes from a call to 'IssueCreationHelperBean'. The 'createFieldScreenRenderer' method is used. It takes a User and (paradoxically) an Issue as parameters.

The Issue creation process begins with a completely blank Issue. It has; no type, no key, no project and no field values. Before 'createFieldScreenRenderer' is called the Issue should have had some of its fields set:




The 'IssueCreationHelper' ('IssueCreationHelperImpl') passes the call to 'FieldscreenRendererFactoryImpl' via its 'getFieldScreenRenderer' method. The FieldscreenRendererFactory also takes an Issue and User as parameters but adds an IssueOperation. The operation is IssueOperations.CREATE_... for this action. The method creates a new 'FieldscreenRendererImpl' using the passed in parameters and adding a FieldManager and a FieldLayoutManager. (Other implementations exist such as BulkFieldScreenRendererImpl)

FieldscreenRenderImpl only uses a subset of the Issue interface's methods:

- getGenericValue (null is a valid value)
- getProject
- getIssueType
- isCreated

The issue is also used for permission checks, but the checks themselves stick to the four methods above as well.

As there is no contract for the part created Issue there is no gaurentee that in the future the Issue will always have these methods (although it seems likely) and that FieldscreenRenderer implementations will never call other methods.

Calls needed to display the sub-task's fields

# Action Validation

## CreateSubtaskIssueDetails extends CreateIssueDetails

Checks:

- ParentIssueId Is set and references a valid issue
- Parent's doValidate

Sets:

- Creates the Issue using IssueManager.getIssueObject(parentIssueId)
- ParentId appears to need doing separately to passing it as a parameter for Issue creation
- Project probably unnecessary as the call to super also does this
- IssueType probably unnecessary as the call to super also does this

## CreateIssueDetails extends CreateIssue

Checks:

- The license
- If it has already failed fail: returns without re-testing
- Its parent's doValidation method fail: for-each error in getErrors() calls addErrorMessage
- IssueCreationHelperBean.validate
- Check if there are attachments fail: Adds an error message 'upload lost due to errors'

Sets:

- Project
- IssueType

## CreateIssue extends AbstractIssueSelectAction

Checks:

- ProjectId Uses IssueCreationHelperBean.validateProjectId

- IssueType Uses IssueCreationHelperBean.validateIssueType

It does not call its parent's doValidate method.

### IssueCreationHelperBean

The implementation of IssueCreationHelperBean doesn't carryout the validation itself, it uses the Field object to carryout the validation. When called as 'validate' it uses the FieldscreenRenderer to identify which fields were in the form and then passes control to each of them in turn.

For each field it calls:

- populateFromParams
- validateParams

## Bulk Operation Actions

Abstract classes exist for Bulk Operation Actions to extend. They are not required however; struts interacts with the objects by looking up methods, via reflection, it ignores inheritance and/or interfaces. The utility methods provided by the abstract classes are only wrappers to static calls to 'BulkEditBean' and so can easily be re-created or replaced.

### AbstractBulkOperationDetailsAction extends AbstractBulkOperationAction

Adds

```
<String>    doDetails()
<String>    doDetailsValidation()
<String>    doPerform()
```

Presumably these are for the 'details' and 'perform' commands. Struts uses reflection to find these methods, it doesn't check the inheritence/interface of the Action just that a method with this name is found. It should be possible to implement these methods without extending AbstractBulkOperationDetailsAction without changing struts behaviour.

### AbstractBulkOperationAction extends IssueActionSupport

Much of this class is for the select operation screen. The 'BulkEditBean' handling methods are the only ones that are important to an implementation of AbstractBulkOperationDetailsAction.

- getBulkEditBean Identical to getRootBulkEditBean
- clearBulkEditBean Required on cancel
- getRootBulkEditBean get's the BulkEditBean object from the session

The methods are just wrappers of methods that already exist on BulkEditBean:

- clearBulkEditBean BulkEditBean.removeFromSession
- getRootBulkEditBean BulkEditBean.getFromSession

It's not clear how the static methods determine which session to get, possibly a look-up based on the thread?

# Implementation

Extending 'CreateSubtaskIssueDetails' makes more sense than extending

'AbstractBulkOperationDetailsAction'. Doing this requires remembering to add 'doDetails', 'doDetailsValidation' and 'doPerform' and using direct calls to BulkEditBean but these are the only inconveniences. Attempting to use the methods of 'CreateSubtaskIssueDetails' without being a sub-class of it would be significantly more complex.

## Single Issue Creation

'CreateSubtaskIssueDetails' and 'CreateIssueDetails' act on a single Issue object.

During validation:

The FieldValuesHolder (map) is populated during validation via the 'populateFromParams' and 'validateParams' methods. An Issue object is created and some of its fields are directly set (parentId, project, IssueType).

During execution:

The data from the 'FieldValuesHolder' is added to the Issue object. IssueCreationHelperBean organizes the transfers, in the 'updateIssueFromFieldValuesHolder' method. It uses a FieldscreenRenderer to decide which Fields need to be transferred. The transfers are carried out by the Fields through calls to the 'updateIssue' method.

Once the IssueObject has been prepared the createIssue call can be made. IssueManager.createIssue only takes two parameters, a User and a Map. However the Map's contents are much like parameters to the call. The Map contains:

- issue The populated, but not yet stored, Issue Object
- pkey <String> The project key

Workflow params:

- WorkflowFunctionUtils.ORIGINAL_ISSUE_KEY ?
- submitbutton Only set if an auxiliary Submit Button was used (workflow related)

The IssueManager createIssue method doesn't directly create the Issue. It passes the parameters map to the WorkflowManager's createIssue method.

The WorkflowManager extracts the Issue and uses the ProjectId and the IssueTypeId to determine which workflow should be used. It makes the workflow and then passes the parameters map into the workflow's initilizer.

OS Workflow takes over control. All of the workflows have a post-function in the first step called 'IssueCreateFunction'. This function unpacks the Issue; sets defaults, if needed, for some of the system fields:

- Created now
- Updated now
- Votes 0

The following system fields are always regenerated:

- key
- workflowId
- statusId

The Issue is then, finally, written to the backing store.

The modified values are moved to become the current values and the list of modified values is cleared.

## Multiple Sub-tasks

### Validation

All of the sub-tasks share the same project and issue-type. These therefore only need testing once, which is what is currently done.

ParentId has as many values as there are sub-tasks to create. The values for ParentId are taken from the Issues selected in the BulkEditBean. If the bean validates its entries then there is no need to re-validate them.

#### Conclusion

Validation should therefore only require a call to super.

### Perform / Execute

The only difference between the Sub-tasks created will be their 'parentId' and 'key' fields.

The Issue object is modified as it passes through the creation process. This means that if a reference of the issue were taken before the creation process the Issue it pointed to would haven been modified into the created Issue. A copy of the FieldValuesHolder could be taken at the start of the 'doExecute' method and a new empty Issue generated and populated for each new sub-task.

The danger of simply using one Issue object and modifying it between each sub-task creation is that if a reference is held to it elsewhere its key is changing, effectively altering the sub-task that is being referenced.

#### Conclusion

The creation action will be very similar to that of CreateIssueAction, except with a loop enclosing the transfer of values to the issue and the creation of the issue. The loop should also include the creation of a new Issue object and setting its project, type and parent.

# Velocity ImportTools

Velocity's tools package is installed as part of JIRA's libraries. One of the tools, the import tool, would allow insertion of processed JSPs into the template. Despite the inclusion of the tools JIRA does not have them setup correctly. '$import' can be used to reference the instance of the tool that Velocity has automatically placed in the context. The object accessed this way has not been initialized though and does not have the required ViewContext.

## Including a JSP in a velocity template

`org.apache.velocity.tools.view.tools.ImportTool`
The import tool needs to be added to the velocity environment. It can then be used in the velocity template. The format is:

```
$import.read("/test.jsp")
```

## ViewContext

ViewContext is an interface. It wraps a number of JSP/Servlet specific Objects together with some Velocity ones. Its intended use is as a transport for these Objects to 'VelocityTool' implementations. It contains 'get' methods for:

- Request
- Response
- ServletContext
- VelocityContext
- VelocityEngine

It also contains a method called 'getAttribute' that searches the request, session and application scope(s) for a value.

## ChainedContext (implements ViewContext)

Javadoc:

```
Velocity context implementation specific to the Servlet environment.
It provides the following special features:
    • puts the request, response, session, and servlet context objects into the Velocity
      context for direct access, and keeps them read-only
    • supports a read-only toolbox of view tools
    • auto-searches servlet request attributes, session attributes and servlet context
      attribues for objects
The ChainedContext.internalGet(String key) method implements the following search order
for objects:
    1. toolbox
    2. servlet request, servlet response, servlet session, servlet context
    3. local hashtable of objects (traditional use)
    4. servlet request attribues, servlet session attribute, servlet context attributes
The purpose of this class is to make it easy for web designer to work with Java Servlet
based web applications.
```

### Constructors

The constructor takes the following parameters, which closely match the get methods provided by the ViewContext interface:

- org.apache.velocity.app.**VelocityEngine**
- javax.servlet.http.**HttpServletRequest**
- javax.servlet.http.**HttpServletResponse**
- javax.servlet.**ServletContext** (This is passed into the constructor of Servlets)
- org.apache.velocity.context.**Context** (optional)

Deprecated versions of the constructor exist that do not require a VelocityEngine.

The actual VelocityEngine that JIRA is using is inaccessible, it is stored as a private variable on an Object that is referenced via a Proxy. The VelocityEngine appears to only be used for the context to register itself and so isn't needed for the context to work. To avoid a null pointer exception a new VelocityEngine is created for the call. No reference is kept, this allows it to be garbage collected as soon as the context has finished with it.

# Notes

I think that if the sub-task has required fields that are not set in the quick create form then the full form will be displayed next. [Needs testing, but could help]

If so a JS field that is required but hides itself could be used to ensure that the form was always displayed.

This still leaves restricting the options on the sub-task type drop-down menu but it is an improvement.