Version Control System Plugin (old style - prior to 4.5)

In TeamCity a plugin for Version Control System (VCS) is seen as an jetbrains.buildServer.vcs.VcsSupport instance. All VCS plugins must extend this class.

VCS plugin has a server side part and an optional agent side part. The server side part of a VCS plugin should support the following mandatory operations:

- collecting changes between versions
- building of a patch from version to version
- get content of a file (for web diff, duplicates finder, and some other places)

There are also optional parts:

- labeling / tagging
- personal builds

Personal builds require corresponding support in IDE. Chances are we will eliminate this dependency in the future.

You can use source code of the existing VCS plugins as a reference, for example:

- http://www.jetbrains.net/confluence/display/TW/Mercurial
- http://www.jetbrains.net/confluence/display/TW/AccuRev

Before digging into the VCS plugin development details it's important to understand the basic notions such as Version, Modification, Change, Patch, Checkout Rule, which are explained below.

Version

A Version is unambiguous representation of a particular snapshot within a repository pointed at by a VCS Root. The current version represents the head revision at the moment of obtaining.

The current version& is obtained from the VcsSupport#getCurrentVersion(VcsRoot). The version here is arbitrary text. It can be transaction number, revision number, date and so on. Usually format of the version depends on a version control system, the only requirement which comes from TeamCity - it should be possible to sort changes by version in order of their appearance (see VcsSupport#getVersionComparator() method).

Version is used in several places:

- for changes collecting
- for patch construction
- when content of the file is retrieved from the repository
- for labeling / tagging

TeamCity does not show Versions in the UI directly. For UI, TeamCity converts a Version to its display name using VcsSupport# getVersionDisplayName(String, VcsRoot).

Collecting Changes

A Change is an atomic modification of a single file within a source repository. In other words, a Change corresponds to a single increment of the file version.

A Modification is a set of Changes made by some user at a certain moment. It most closely corresponds to a single checkin transaction, when a user commits all his modifications made locally to the central repository. A Modification also contains the Version of the VCS Root right after the corresponding Changes have been applied.

TeamCity server polls VCS for changes on a regular basis. A VCS plugin is responsible for collecting information about Changes (grouped into Modifications) between two versions.

Once a VCS Root is created the first action performed on it is determining the current Version (VcsSupport#getCurrentVersion(VcsRoot)). This value is stored and used during the next checking for changes as the "from" Version (VcsSupport#collectBuildC

hanges(VcsRoot, String, String, CheckoutRules)). The current Version is obtained again to be used as the "to" Version. The Modifications collected are then shown as pending changes for corresponding build configurations. After the checking for changes interval passes the server requests for next portion of changes, but this time the "from" Version is replaced with the previous "current" Version. And so on.

Obtaining the current Vesion may be an expensive operation for some version control systems. In this case some optimization can be done by implemeting interface CurrentVersionIsExpensiveVcsSupport. Its method CurrentVersionIsExpensiveVcsSupport #collectBuildChanges(VcsRoot, String, CheckoutRules) takes only "from" Version assuming that the changes are to be collected for the head snapshot. In this case TeamCity will look for the Modification with the greatest Version in the returned Modifications and take it as the "from" parameter for the next checking cycle. If you implement CurrentVersionIsExpensiveVcsS upport, the you can leave method VcsSupport#collectBuildChanges(VcsRoot, String, String, CheckoutRules) not implemented.

Patch Construction

A Patch is the set of all modifications of a VCS Root made between two arbitrary Versions packed into a single unit. With Patches there is no need to retrieve all the sources from the repository each time a build starts. Patches are sent to agents where they are applied to the checkout directory. Patches in TeamCity have their own format, and should be constructed using jetbrains.buildServer.vcs.patches.PatchBuilder .

When a build is about to start, the server determines for which Versions the patch is to be constructed and passes them to Vcs Support#buildPatch(VcsRoot, String,String, PatchBuilder, CheckoutRules).

There are two types of patch: clean patch (if fromVersion is null) and incremental patch (if fromVersion is provided). Clean patch is just an export of files on the specified version, while incremental patch is a more complex thing. To create incremental patch you need to determine the difference between two snapshots including files and directories creations/deletions.

Checkout Rules

Checkout rules allow to map path in repository to another path on agent or to exclude some parts of repository, read more.

Checkout rules consist of include and exclude rules. Include rule can have "from" and "to" parts ("to" part allows to map path in repository to another path on agent). Mapping is performed by TeamCity itself and VCS plugin should not worry about it. However a VCS plugin can use checkout rules to speedup changes retrieval and patch building since checkout rules usually narrow a VCS Root to some its subset.

In most cases it is simpler to collect changes or build patch separately by each include rule, for this VCS plugin can implement interface jetbrains.buildServer.CollectChangesByIncludeRule (as well as jetbrains.buildServer.vcs.BuildPatchByIncludeRule) and use jetbrains.buildServer.vcs.VcsSupportUtil as shown below:

```
public List<ModificationData> collectBuildChanges(VcsRoot root, String fromVersion, String
currentVersion, CheckoutRules checkoutRules)
  throws VcsException {
  return VcsSupportUtil.collectBuildChanges(root, fromVersion, currentVersion, checkoutRules, this);
}

public List<ModificationData> collectBuildChanges(VcsRoot root, String fromVersion, String
currentVersion, IncludeRule includeRule)
  throws VcsException {
  ... changes collecting code ...
}
```

And for patch construction:

```
public void buildPatch(VcsRoot root, String fromVersion, String toVersion, PatchBuilder builder,
CheckoutRules checkoutRules)
  throws IOException, VcsException {
  VcsSupportUtil.buildPatch(root, fromVersion, toVersion, builder, checkoutRules, this);
}

public void buildPatch(VcsRoot root, String fromVersion, String toVersion, PatchBuilder builder,
IncludeRule includeRule)
  throws IOException, VcsException {
  ... build patch code ...
}
```

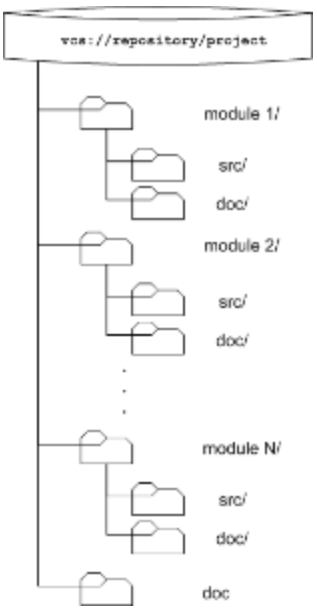
If you want to share data between calls, this approach allows you to do it easily using anonymous classes:

```
public List<ModificationData> collectBuildChanges(VcsRoot root, String fromVersion, String
currentVersion, CheckoutRules checkoutRules)
  throws VcsException {
  final MyConnection conn = obtainConnection(root); // get a connection to the repository
  return VcsSupportUtil.collectBuildChanges(root, fromVersion, currentVersion, checkoutRules, new
CollectChangesByIncludeRule {
    public List<ModificationData> collectBuildChanges(VcsRoot root, String fromVersion, String
currentVersion, IncludeRule includeRule)
      throws VcsException {
      doCollectChange(conn, includeRule); // use the same connection for all calls
    }
  });
}

public List<ModificationData> collectBuildChanges(VcsRoot root, String fromVersion, String
currentVersion, IncludeRule includeRule)
  throws VcsException {
  ... changes collecting code ...
}
```

When using `VcsSupportUtil` it is important to understand how it works with Checkout Rules and paths. Let's consider an example with collecting changes.

Suppose, we have a VCS Root pointing to `vcs://repository/project/`. The project root contains the following directory structure:

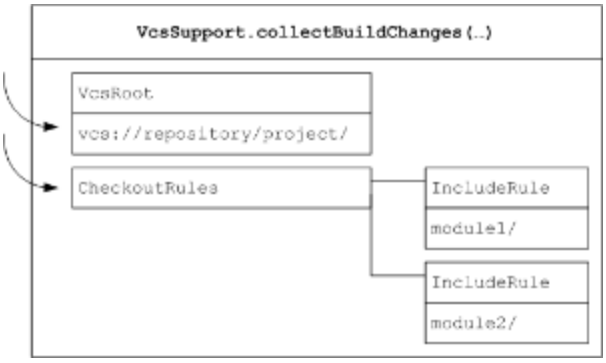


We want to monitor changes only in module1 and module2. Therefore we've configured the following checkout rules:

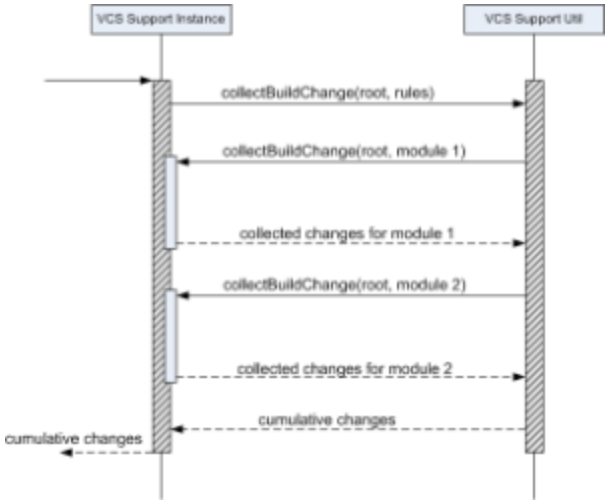
```
\+:module1

\+:module2
```

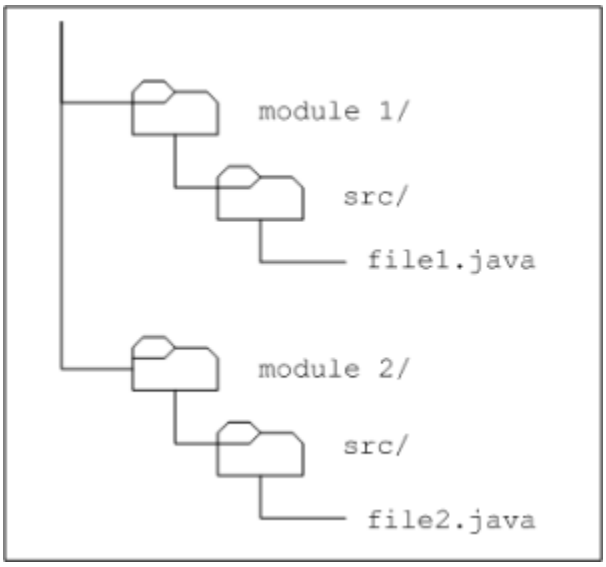
When `collectBuildChanges(...)` is invoked it will receive a `VcsRoot` instance that corresponds to `vcs://repository/project/|| and a {{CheckoutRules` instance with two `IncludeRules` — one for "module1" and the other for "module2".



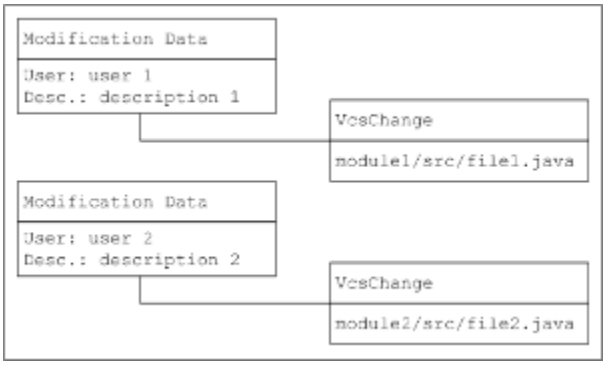
If `collectBuildChanges(...)` utilizes `VcsSupportUtil.collectBuildChanges(...)` it transforms the invocation into two separate calls of `CollectChangesByIncludeRule.collectBuildChange(...)`. If you have implemented `CollectChangesByIncludeRule` in the way described in the listing above you will have the following interaction.



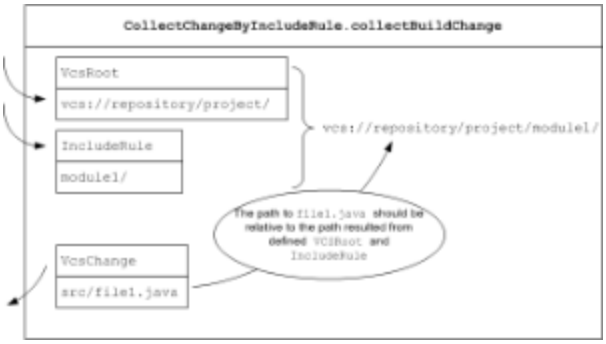
Now let's assume we've got a couple of changes in our sample repository, made by different users.



The collection of ModificationData returned by VcsSupport.collectBuildChanges(...) should then be like this:



But this is not a simple union of collections, returned by two calls of CollectChangesByIncludeRule.collectBuildChange(...). To see why let's have a closer look at the first calls.

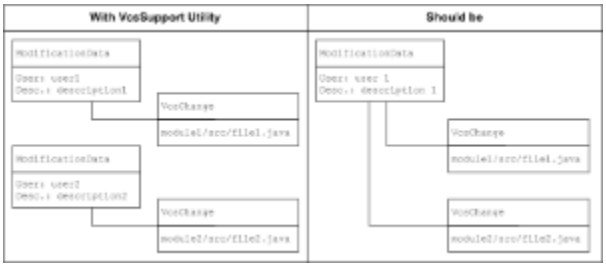


As you can see the paths in the resulting change must be relative to the path presented by the include rule, rather than the path in the VCS Root.

Then after collecting all changes for all the include rules `VcsSupportUtil` transforms the collected paths to be relative to the VCS Root's path.

Although being quite simple `VcsSupportUtil.collectBuildChanges(...)` has the following limitations.

Assume both changes in the example above are done by the same user within the same commit transaction. Logically, both corresponding `VcsChange` objects should be included into the same `ModificationData` instance. However, it's not true if you utilize `VcsSupportUtil.collectBuildChanges(...)`, since a separate call is made for each include rule.



Changes corresponding to different include rules cannot be aggregated under the same ModificationData instance even if they logically relate to the same commit transaction. This means a user will see these changes in separate change lists, which may be confusing. Experience shows that it's not very common situation when a user commits to directories monitored with different include rules. However if the duplication is extremely undesirable an implementation should not utilize `VcsSupportUtil.collectBuildChanges(...)` and control `CheckoutRules` itself.

Another limitation is complete ignorance of exclude rules. As it said before this doesn't cause showing unneeded information in the UI. So an implementation can safely use `VcsSupportUtil.collectBuildChanges(...)` if this ignorance doesn't lead to significant performance problems. However, If an implementer beleives the change collection speed can be significantly improved by taking into account include rules, the implementation must handle exclude rules itself.

All above is applicable to building patches using `VcsSupportUtil.buildPatch(...)` including the path relativity aspect.

Registering In TeamCity

During the server startup all VCS plugins are required to register themselves in the VCS Manager (jetbrains.buildServer.vcs.VcsManager). A VCS plugin can receive the VcsManager instance using Spring injection:

```
class SomeVcsSupport implements VcsSupport {
...
  public SomeVcsSupport(VcsManager manager) {
    manager.registerVcsSupport(this);
  }
...
```

</p>

Server side caches

By default, server caches clean patches created by VCS plugins, because on large repositories clean patch construction can take significant time. If clean patches created by your VCS plugin must not be cached, you should return true from the method VcsSupport#ignoreServerCachesFor(VcsRoot).

Agent side checkout

Agent part of VCS plugin is optional; if it is provided then checkout can also be performed on the agent itself. This kind of checkout usually works faster but it may require additional configuration efforts, for example, if VCS plugin uses command line client then this client must be installed on all of the agents.

To create agent side checkout, implement jetbrains.buildServer.agent.vcs.CheckoutOnAgentVcsSupport in the agent part of the plugin. Also server side part of your plugin must implement jetbrains.buildServer.AgentSideCheckoutAbility interface.

See Also:

- Extending TeamCity: Developing TeamCity Plugins | Typical Plugins