Version Control System Plugin

Overview

In TeamCity a plugin for Version Control System (VCS) is seen as a set of interface implementations grouped together by instances of

jetbrains.buildServer.vcs.VcsSupportContext (server-side part) and jetbrains.buildServer.agent.vcs.AgentVcsSupportContext (agent-side part).

The server-side part of a VCS plugin is responsible the following major 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, which require corresponding support in IDE. This dependency may be eliminated in the future.

The agent-side part is optional and only responsible for checking out and updating project sources on agents. In contrast to server-side checkout it offers a traditional approach to interacting between a CI system and VCS – when source code is checked out into the same location where it's built. For pros & cons of both solutions see VCS Checkout Mode.

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

- Git plug-in
- Mercurial plug-in

For more information on TeamCity plugins, please refer to the TeamCity Plugins section.

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

Basic Terms

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 taken by calling jetbrains.buildServer.vcs.CollectSingleStatePolicy#getCurrentVersion(jetbrains.buildServer.vcs.VcsRoot). The version here is an arbitrary text. It can be a representation of a transaction number, a revision number, a date, whatever suitable enough for getting a source snapshot in a particular VCS. 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 happening (see jetbrains.buildServer.vcs.VcsSupportConfig#getVersionComparator()).
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 jetbrains.buildServer.vcs.VcsSupportConfig#getVersionDisplayName(String,jetbrains.buildServer.vcs.VcsRoot).

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 a file version.

A Modification is a set of changes made by some user at a certain time interval. It most closely corresponds to a single checkin transaction (commit), 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.

A collection of Modifications is what TeamCity expects as a result when asking a VCS plugin for changes.

A Patch is a set of operations to convert the directory state from one modification to another (e.g. change/add/remove file, add/remove directory).

A Checkout Rule is a way of changing default file layout.

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

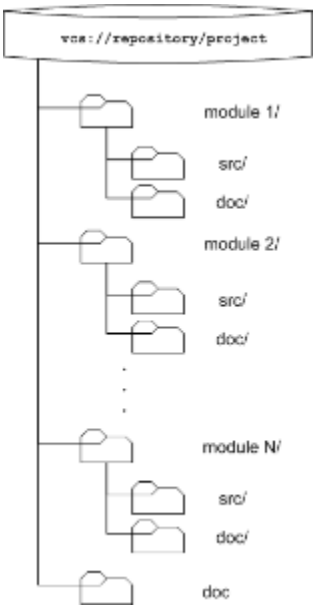
Checkout rules consist of include and exclude rules. Include rule can have "from" and "to" parts ("to" part allows to map path in repository to another path on agent). Mapping is performed by TeamCity itself and VCS plugin should not worry about it. However a VCS plugin can use checkout rules to speedup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

Server-Side Part

Patch Building and Change Collecting Policies

When implementing include rule policies 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:

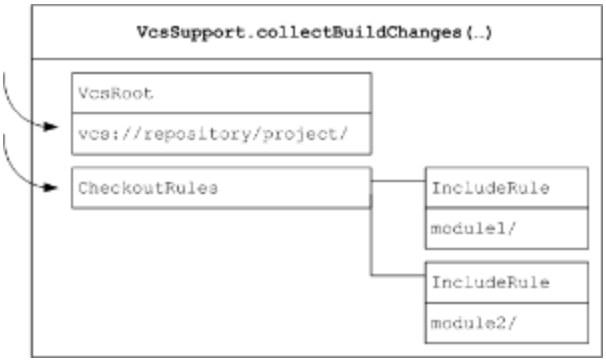


We want to monitor changes only in `module1` and `module2`. Therefore we've configured the following checkout rules:

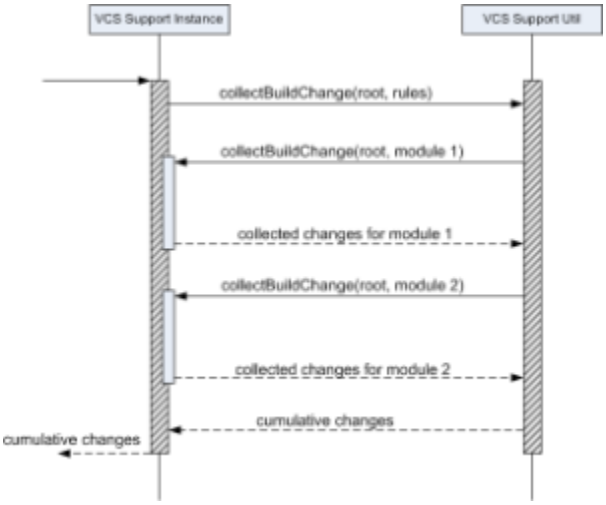
```
\+:module1

\+:module2
```

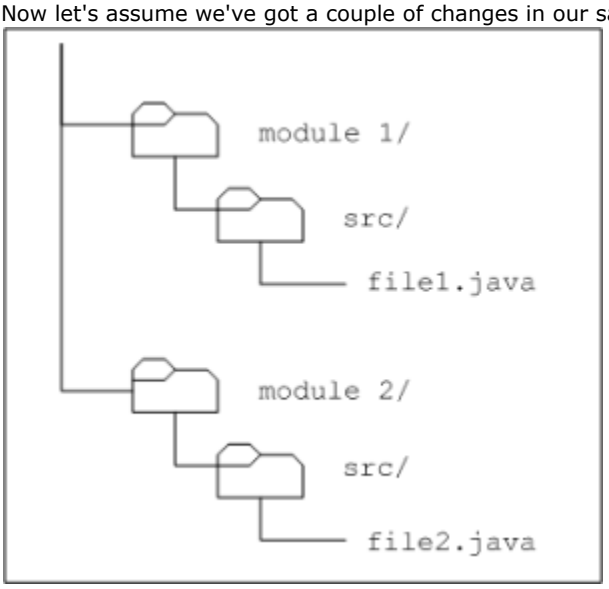
When `collectBuildChanges(...)` is invoked it will receive a `VcsRoot` instance that corresponds to `vcs://repository/project/` and a `CheckoutRules` instance with two `IncludeRules` — one for "module1" and the other for "module2".



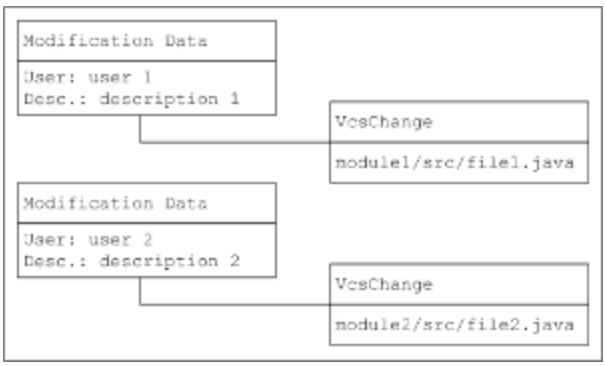
If `collectBuildChanges(...)` utilizes `VcsSupportUtil.collectBuildChanges(...)` it transforms the invocation into two separate calls of `CollectChangesByIncludeRule.collectBuildChange(...)`. If you have implemented `CollectChangesByIncludeRule` in the way described in the listing above you will have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



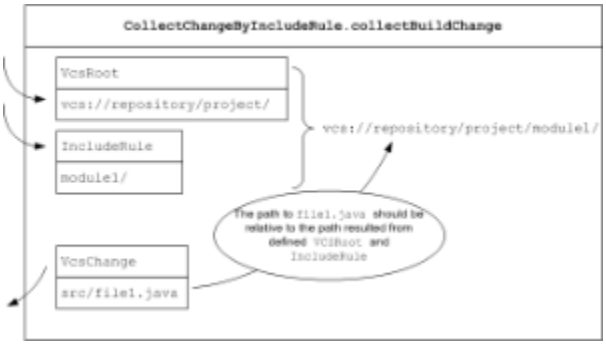
Now let's assume we've got a couple of changes in our sample repository, made by different users.



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.

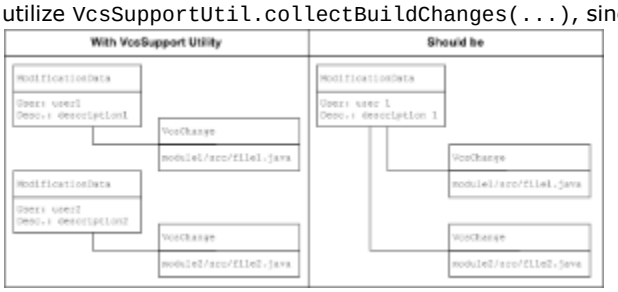


As you can see the paths in the resulting change must be relative to the path presented by the include rule, rather than the path in the VCS Root.

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 believes 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.

Server-Side Caching

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

Agent-Side Part

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 enable agent-side checkout, be sure to include jetbrains.buildServer.agent.vcs.AgentVcsSupportContext into agent plugin part and also enable agent-side checkout via jetbrains.buildServer.vcs.VcsSupportConfig#isAgentSideCheckoutAvailable().