



# Plastic SCM

## Component Oriented Development

Whether it is decided or not from day one it doesn't really matter, at a certain moment in time you will decide to split your project into different components.

How will you reflect these changes in your SCM? Can your version control system help you managing components?

Read how Plastic can help you achieving these goals.

Component based software development is not only good for boosting reuse in your project, it also has some other benefits like simplifying project management. Yes, splitting up your software depending on the number of developers or their abilities doesn't look like a good idea, but correctly dividing the whole system into meaningful modules looks better and at a certain point in time this will help on splitting up responsibilities.

Some projects will face component division from day one (creating different libraries which will follow their own release cycles), and will even have different groups in charge of each of the components. In other projects software is started as a whole (and it doesn't mean it is not correctly designed but it is all built and released together), but time and the ever growing feature list will force the team to divide it into independent pieces, and sometimes the team itself will be split to work on the different parts.

### What's a component?

There are several definitions for components in the software industry, but we're talking about version control and configuration management so for us a component will be a group of files and directories which are built together, sharing the same versioning schema and maintained together.

So if a software project is built together and has the same versioning it will be the same component for us, no matter whether it has a couple or a thousand classes.

Figure 1 shows a sample project structure:

- A *bin* directory to hold build output
- A *lib* directory with external dependencies
- A *doc* directory to place analysis and design documents
- A *build* directory which contains the build scripts
- A *src* directory for source code

The typical structure described can vary depending on the project's needs or even the programming language but they will all share a very similar project tree.

Inside the *src* directory the sample project code has been divided into (it could be a business application, for instance):

- *auth* directory for authentication
- *dbbackend* for the database abstraction layer
- *customers* for customer management

- *sales* for the code handling sale management

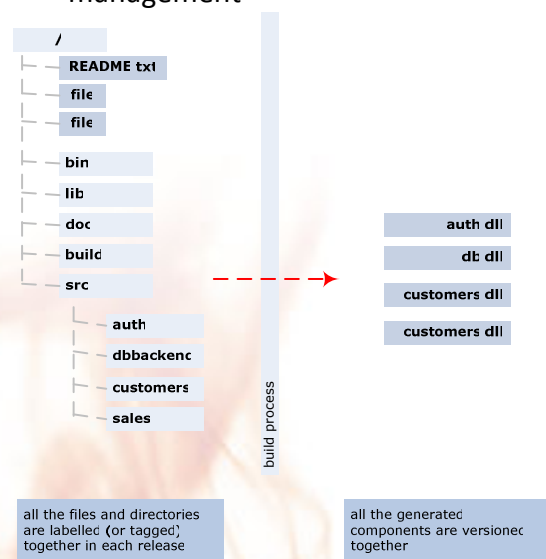


Figure 1. Sample project tree

Inside the different source directories there will be files and the classes they implement.

The structure would be different in a typical Java system, with directories under *src* resembling the package hierarchy.

The important point here is that when the system is built (compiled, unit tested and so on) several binary libraries (or assemblies, or packages, depending on the actual technology used) will be generated. Each of them can be considered as a *component* but from the version control point of view they are all released together and they will all be labeled together, so they're just a single *component*.

For many projects a scenario like the described will end up having *build 1*, *build 2* ... *build 103* and so on, the elements (auth.dll,

db.dll, customers.dll and son on) will not be numbered differently.

On the other hand (and specially following agile methods, remember *collective code ownership*) it makes sense that a developer working on a certain task makes changes to source code in different *subcomponents*. Maybe a bug fix will touch files under both *auth* and *dbbackend*, and it won't be a problem. Conceptually the software under development is considered a unit (with *sub elements*) and deployed and distributed as such.

### To split or not to split...

In general I would strongly recommend following the KISS<sup>1</sup> principle, which implies dealing with the project as a single unit if it is possible. Of course it should be designed to be modular, split into *assemblies* and *packages*, writing the right classes and so on, but if possible, from the version control point of view it should be treated as a single unit: branch it all together, label it all together.

But there will be situations where things can't be that simple: you will have deployment constraints forcing you not to release the whole binaries for a single change, but just the affected binary module. Another case will be that the system is big enough to be developed by a number of teams and they need to be coordinated somehow, or even that you're in charge of different libraries which are used by different projects inside the organization.

If this is the case, you'll need to reflect your component model into your SCM solution, and that's exactly what we'll be talking about in the following topics.

### SCM components

Consider **Figure 2** in which the project structure defined in **Figure 1** is divided into three components: *auth.dll*, *db.dll* and the business rules component which groups

together the *customers* and *sales* source trees.

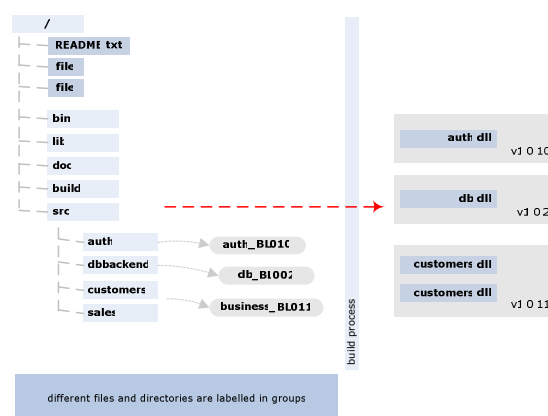


Figure 2. SCM components defined

As mentioned above, there can be several reasons to end up with such configuration: maybe *auth* and *db* components need to be used by other projects following different schedules, maybe there are three groups working on the different modules and you need to ensure that they get in synch regularly but always using stable releases from each other so they won't waste time fixing someone else's bugs.

The SCM system will have to help the CM and project managers to define and enforce the desired component structure and their release cycles.

### Repositories and components

Plastic SCM supports the concept of *multirepository* development which means that the source tree can be *split* into different repositories and still be presented to the developer like only one.

From a CM or project manager point of view it will be necessary to define when the project needs to be set up using one or several repositories.

Again, there is no general rule of thumb, but trying to keep things simple will help: Plastic SCM lets you create several repositories but it doesn't mean that you have to do it.

If you divided the system into components to better coordinate different groups' efforts and define clear release interactions, but the components won't be used by more than a project or product, they should be kept in a single repository.

<sup>1</sup>[http://en.wikipedia.org/wiki/Keep\\_it\\_simple\\_stupid](http://en.wikipedia.org/wiki/Keep_it_simple_stupid)

If the components are used by more than one project or product, each of them should be created on its own repository. **Figure 3** shows an option in which all three components are just placed into the same repository. They will still be released separately and labeled independently but stored in the same repository.

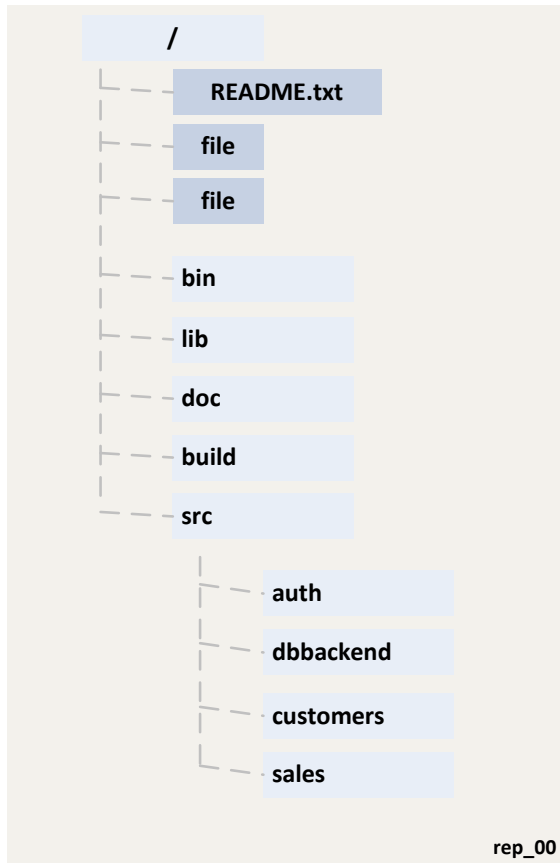


Figure 3. All components inside the same repository

**Figure 4** divides the components into three repositories.

The *rep\_00* contains the main project structure plus the business rules component.

*rep\_db* stores the *dbbackend* component source code.

*rep\_auth* the *auth* source tree.

The project could be divided into even more components separating the *business logic* from the main project structure. The following topics will describe how Plastic can be used to define such a repository structure and how to set up developer's workspaces to seamlessly work with these configurations.

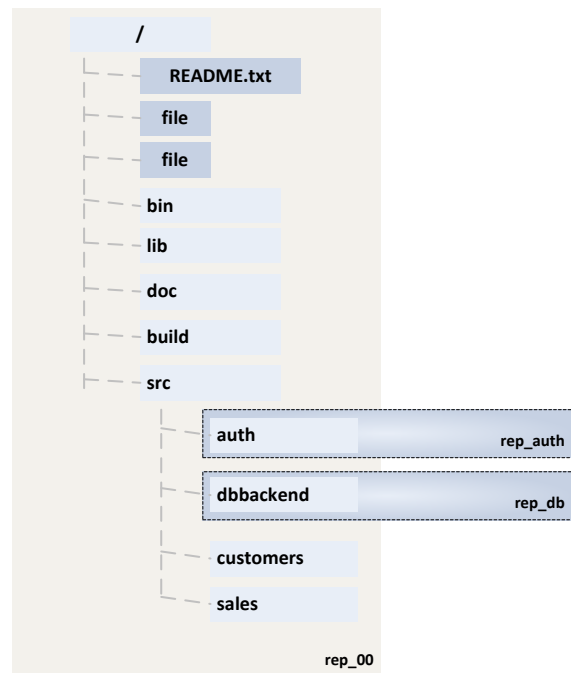


Figure 4. Components stored into three repositories

### Single repository set up

In this scenario different development groups will work on the *business logic*, *authentication*, and *database* components. They will all share the basic directory structure (most likely creating documentation under the same trees) but with different configurations in order to be able to always work against a stable code base.

Let's start with the group in charge of the business rules: At a given moment in time they'll be working on their *baseline 10* in order to create *baseline 11*. Imagine that a developer has to work on task *logic1492* correctly configured to compile against the right sources.

The conventional selector to achieve this would be:

```
rep "rep_00"
  path "/"
    branch "/main/logic1492" label "baseline10"
    checkout "/main/logic1492"
```

But this implies that *all the code* under the project's root directory "/" will be taken from *baseline 10* and branched to *logic1492* if changed. That's not the correct scenario anymore because now the subdirectory trees *auth* and *dbbackend* are managed independently.

Both components could have been *labeled* with *baseline 10* when the baseline was created so it could work, but more often than not the configuration would be something like: work against *baseline 10* for the business logic but against *auth\_BL010* for *auth* and *db\_BL002* for *dbbackend* tree.

In order to achieve it, the following selector will be required:

```
rep "rep_00"
  path "/src/auth"
  label "auth_BL010"
  path "/src/dbbackend"
  label "db_BL002"
  path "/"
  branch "/main/logic1492" label "baseline10"
  checkout "/main/logic1492"
```

Notice that the *selector rules* for paths */src/auth* and */src/dbbackend* don't include a *checkout* rule, which means that they can't be modified by using this selector.

Suppose now that a developer on the *auth* group needs to work on task *aut101*, but he needs to load *baseline 09* from the rest of the code in order to reproduce a specific bug.

He would set up the following selector:

```
rep "rep_00"
  path "/src/auth"
  branch "/main/aut101" label "auth_BL010"
  checkout "/main/aut101"
  path "/"
  label "baseline09"
```

Now a third developer from the *db* group needs to perform a refactor on the *db* internal code. He doesn't want to load the entire project's code into his workspace, just the *db* tree. He will use a selector like the following:

```
rep "rep_00"
  path "/src/dbbackend"
  branch "/main/db099" label "db_BL002"
  checkout "/main/db099"
  path "/src" norecursive
  label "baseline010"
  path "/" norecursive
  label "baseline010"
```

Please note that the selector defines rules to load the entire path until the *dbbackend* tree with the two last rules but adding the *norecursive* option which means that the directory itself is loaded but none of its entries is downloaded.

## Multi-repository set up

But, what if we want to work with a scenario like the one defined by **Figure 4**?

Then developers will have to use the selector's ability to load more than one repository at the same time.

Now the *rep\_auth* will contain the code to be mounted under the */src/auth* directory and *rep\_db* the code for */src/dbbackend*.

Suppose you need to *mount* baseline *auth\_BL010* for *auth* and *db\_BL002* for *db*.

The selector would look like the following:

```
rep "rep_auth" mount ""/src/auth"
  path "/"
  label "auth_BL010"
rep "rep_db" mount ""/src/dbbackend"
  path "/"
  label "db_BL002"
rep "rep_00"
  path "/"
  branch "/main/logic1492" label "baseline10"
  checkout "/main/logic1492"
```

A developer working just on *auth* code would just use the *auth* rep:

```
rep "rep_aux"
  path "/"
  branch "/main/aut101" label "auth_BL010"
  checkout "/main/aut101"
```

Selectors can be defined in order to work on several branches on different repositories at the same time, or even on different branches inside the same repository, depending on the project's needs.

## Wrapping up

Plastic allows really flexible project configurations through *selectors*. At the beginning many users consider selectors as just a way to load old revisions in certain circumstances but it is actually quite different: selectors allow full project reorganization and configuration to achieve different goals.