Structural Search and Replace: What, Why, and How-to

Maxim Mossienko, JetBrains

Imagine that we have a large source code-base that we need to browse or modify it. For instance, we might want to use a library and find out how it works, or we might need to get acquainted with existing code and to modify it. Yet another example is that a new JDK becomes available and we are keen to see the changes in the standard Java libraries and so on. Conventional tools like find and replace text may not completely address these goals because when we use them, it is easy to find or replace too much or too little. Of course, if someone already knows the source code well, then using the whole words option and regular expressions may help make our find-and-replace queries smarter.

The problem of the conventional approach, even with regular expressions, is that they just do not know anything about the syntax and semantics of the source code we are using. This is why we combined the search-and-replace feature with knowledge about the source code, producing the Structural Search and Replace (SSR) feature.

Important notes about the user interface and how SSR works

Code templates

The user interface of SSR (Figure 1 and 2) was made as close to conventional search-and-replace as possible. We still need to enter a search string and maybe a replacement string, and specify the case sensitive option. However the similarities end at this point. The search and replace strings in SSR will be, in fact, the code fragments (templates) we would like to find or replace. Any template entered should be a well formed Java construction of one of the following types:

- An expression, like `ProcessCancelledException()`
- A statement or sequence of statements, like: `a = b;`
- A class, e.g. `class A implements B {}`
- A comment or javadoc comment, e.g. `/** @beaninfo */`

Note: *The Copy existing templates button allows you to quickly pick up one of many pre-built Java construction templates (class, methods, ifs, etc) and user defined templates(if any), so quite often there is no need to enter code patterns unless there is some selection of existing source code.*

The matching of the template code with source code is accomplished mostly according to Java syntax rules. This implies, for instance, that white spacing of the template and source code is not significant. Certain semantic knowledge is also applied during search, e.g. the order of the class fields, methods or references in an implements list is not significant.

The matching of the first two template types (see the list of 4 types above) is accomplished strictly, i.e. a match is found when an exact occurrence of the code is found. On the other hand, matching of the third and forth template types is done loosely, meaning that a match could have other content not mentioned in the template. For instance, the search template `new Runnable() {}` will find all anonymous `Runnable` instances. The same convenience shorthand is applicable for method bodies. Thus, the following search template will find any `Runnable` with a method called `someMethod`:

```
new Runnable() { void someMethod(); }
```

References to classes, fields, variables, methods, etc. are treated literally (e.g. search template a = b; matches only a = b;) except for a case mentioned below.

Note: *The SSR operates over the concrete syntax trees of the source code and the supplied code templates. Thus, using code templates with errors or applying SSR to source code with errors ("red") could cause unexpected or undesired effects.*

Template variables

Simple code templates allow finding and replacing exact source code. For instance, searching with the template `aaa instanceof String` will always find exactly this code. However, what if we want to find `<any_expression> instanceof String`? To deal with such problems, SSR has template variables, by which I mean 'some source code reference'. Any reference that looks like `$NameOfReference$` is template variable, where NameOfReference is any user defined name permitted by the Java language. For instance, `$arg$` on Figure 2 is such a template variable.

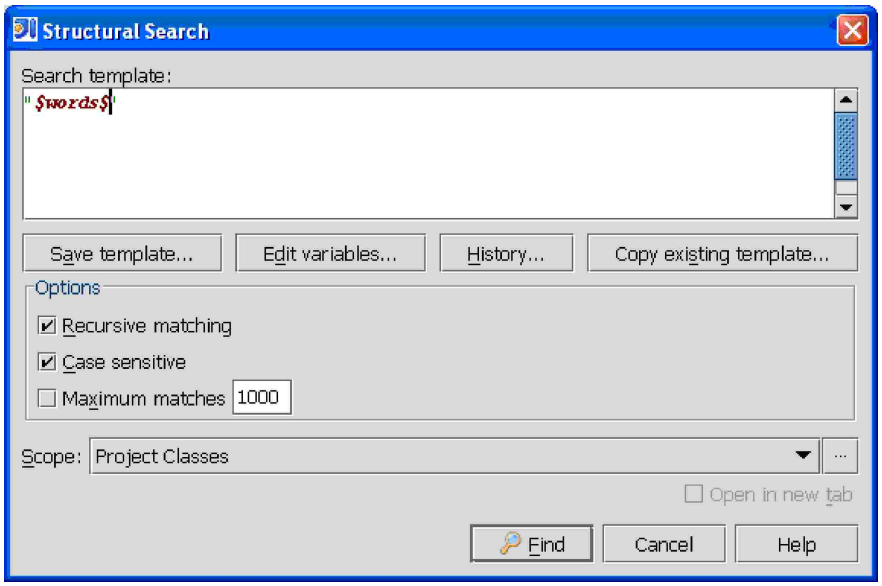

Figure 1 Structural Search dialog

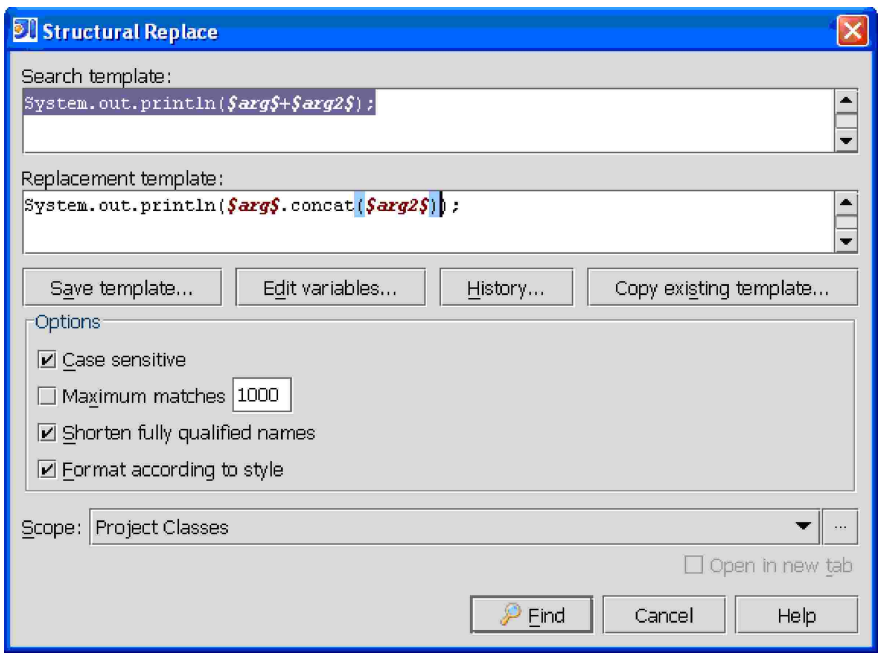

Figure 2 Structural Replace dialog

A template variable matches a source code expression if the expression fits within the user defined constraints (if any) which specify what possible values the template variable may have. Returning to the previous example of finding source code like `<any_expression> instanceof String`, we will simply use the template `$AnyExpression$ instanceof String`. For example, assume that we have the following piece of code in the project:

```
Log.assert(
    context.getProvider().getAccessToken()
    instanceof String );
```

In this case the search result according to the above template will be:

```
context.getProvider().getAccessToken()
instanceof String
```

And the `AnyExpression` template variable will have the match:

```
context.getProvider().getAccessToken()
```

Another important usage of template variables is that whatever expression gets bound to the template variable during the search, it can be used again in the search or replace templates.

Yet another use of template variables is to mark the variable with the 'This variable is the target of the search' option (Figure 3). This means that each match of this particular variable is included in the output of the search. If in our previous example this option was enabled for the `AnyVariable` template variable, the search would have produced the following output:

```
context.getProvider().getAccessToken()
```

By default, this option is off, and the result of the search is each match of the entire search template.

Note: *SSR supports template variables only in certain places, namely Java code references or names, strings, simple comment contents, names of javadoc tags and their values, and name/value pairs after a javadoc tag.*

Introduction to template variable constraints

Consider we want to find all getters declared in the particular class. To find them we could try to use method search template like the one below:

```
class ClassOfInterest { $GetterType$
$GetterName$(); }
```

We will also enable the 'This variable is the target of the search' option for the `GetterName` variable, since we are looking for methods.

However, we also need to specify that we are looking for all getter methods. This is where the template variable constraints come into play. We could specify that a getter method is one that has `GetterName` text which matches the regular expression `get.*` so we enter this expression into the 'Text / regular expression' field for the `GetterName` variable (Figure 3). One more constraint that we need to set is how many occurrences of the particular variable we should match. By default, the minimum occurrences count and maximum occurrences count are set to 1. If

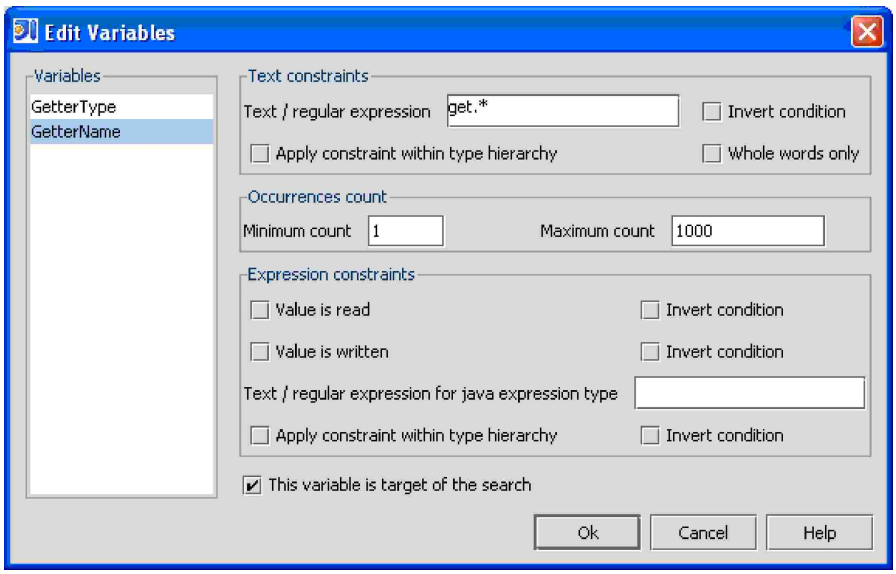

Figure 3 Edit variables dialog

we set the maximum value for `GetterType` and `GetterName` to some larger value (say 1000), our method declaration will be able to match any number of methods from 1 to 1000.

Note: *The constraints for specific template variables can be set by pressing the Edit Variables button (Figure 2).*

Let's explore the 'Text / regular expression' constraint work in more detail. As it is performing a search, the SSR matches template variables to source code expressions. The source code expression corresponds to some source code text. The 'Text / regular expression' constraint means that the source code text must also match a given regular expression for the match to be valid.

Note: *Using regular expressions for text constraints also means that you need to escape the regular expression meta-characters like . () [] - ^ $ \ with one slash (e.g. com\.intellij\.openapi\.editor\.Editor).*

Returning to the previous example, we may want to find all getters declared by a particular class and all its descendants. In order to do so, we will set 'Apply constraint within hierarchy' in the 'Text constraints' group for the GetterName variable. Effectively, the option tells the matcher to go into the type hierarchy when the match is not found locally.

Yet another useful application of this option is to set it for a template variable which represents some type. When checking the constraint, the matcher attempts to compare the source code type's name with the given regular expression. If it fails, the matcher will then check the type's super-class and so on. For instance, search templates `($Type$) $Expr$` or `$Expr$ instanceof $Type$` with a text constraint for the Type variable set to 'PsiElement' with the option 'Apply the constraint within hierarchy' set will search for all casts to or instances of classes that implement the interface `PsiElement`.

Let's cover in more detail the previously mentioned constraints of minimum and maximum occurrences of a template variable (in 'Occurrence count' group, Figure 3). While matching the search template, the template variable could receive not just one value (though this is the default behaviour). but several ones **from the same context**, or receive no value at all. We allow several matches by setting the maximum occurrence count to a value larger than one, and we allow zero matches by setting the minimum occurrence count to 0. For instance, when searching for 'any static method call of a particular class', we may be not interested in the number of parameters passed, so we can use the following search template:

```
Utils.assertTrue($Parameter$);
```

For the `Parameter` template variable, we specify minimum occurrences as 0 and maximum occurrences as some large value (say, 1000). While matching, the Parameter variable will pick up zero or more parameter expressions of the particular call. The commas separating the particular parameters (if any) are not mentioned in the search template, nor taken into account during the match, since their presence has only a textual syntax meaning.

Yet another example of using the occurrence count values is when searching for if statements. The search template looks like following:

```
if ($BooleanExpr$) {
    $ThenStatement$;
} else {
    $ElseStatement$;
}
```

For the `ThenStatement` and `ElseStatem`ent template variables the minimum count constraint is set to 0 and the maximum count is set to Integer.MAX_VALUE.

Note: *This search pattern will also find an `if` without an `else` branch, since the latter is semantically equivalent to an empty else.*

I would like to know more about the source

Finding all descendants of the class

Quite often we need to find all descendants of a particular class. To search for such classes, use the template:

```
class $Clazz$ extends $AnotherClass$ {}
```

As the text constraint for the AnotherClass variable one needs to specify the name of the base class for which we are looking for descendants. Since we are looking for any descendants of the base class we enable 'Apply the constraint within the type hierarchy' option.

Finding all such methods

In certain situations, one needs to look for many different implementations of the same interface method. This can be achieved with the following search pattern:

```
class $a$ {
    public void $show$();
}
```

The text constraint for the `show` variable is 'show', and 'This variable is the target of the search' is enabled.

Finding in literals and comments

Consider that we want to find something in the string literals of our program. For instance, we have some typo in the title of a dialog and we need to find the exact location. The corresponding search template is simple: `"$StringContent$"`, the text constraint for `StringContent` is .*OurTypo.*. However, if we know in advance that the typo is actually a word, then we can use the 'Whole words only' option (with case sensitive search) to perform an indexed source search, which would find it faster. In this case, we set the text constraint as OurTypo. The similar approach is used for searching in the comments, just the search pattern used is different, namely: `// $LiteralContent$`.

Finding specific usages

We may be interested in finding all calls of a particular method on instance variables of a particular descendant type. For instance, find 'equals' calls that are called from instances of the String class. The search pattern would be: `$instance$.equals($argument$)`. Text / regular expression constraint for the instance template variable would be String.

I'm tired of manually editing similar things over and over

Upgrading a library

Quite often, a library's evolution is out of our control. Not all of them preserve backward API compatibility, so you have to change existing code extensively after a major update of a third-party library. For instance, when we were upgrading from Xerces 2.0 to 2.6.2, we needed to replace `DOMInputSourceImpl` with `DOMInputImpl`. This was achieved with the following settings:

Search template: `$InputSourceImpl$`.
Text constraint: 'DOMInputSourceImpl'.
Replacement template: `DOMInputImpl`.

I want to change the classes

Consider we want to change the parent for many classes, e.g. from bare `TestCase` to `OurHomeGrownTestCase`. It is convenient to use Structural Search and Replace to accomplish this with following settings.

Search template:

```
class $TestCase$ extends TestCase {
    $MyClassContent$
}
```

Replacement template:

```
class $TestCase$ extends OurHomeGrownTestCase {
    $MyClassContent$
}
```

The minimum and maximum occurrence counts for the MyClassContent variable should be set 0 and Integer.MAX_VALUE respectively.

Tip: *This way, one could also quickly insert or remove a default implementation of an interface method in many classes when a new method is added or removed from the interface.*

Note: *In case of using the short name of a class in the example above, IDEA will detect that the class is not imported yet, and will suggest automatically adding the necessary import statement where needed.*

As for the replaced class, it may, vice versa, need a removal of the redundant imports. This can be easily achieved by calling the 'Optimize Imports' command.

Heavy refactoring: Static utility method calls -> singleton instance method calls

One of our classes, MakeUtil, was completely refactored during its move to the Open API. All its static methods became instance methods and needed to be called from a singleton instance. The problem was hard due to the fact that we have two such classes in different packages. After modifying the class itself, we needed to update all the usages. Here's how we made the static class update with Structural Search and Replace.

Search template:

```
com.ij.j2ee.MakeUtil.$MethodCall$($Params$)
```

Replace template:

```
com.ij.j2ee.MakeUtil.getInstance().$MethodCall$
($Params$)
```

Constraints:
Minimum and maximum occurrences count for `Params` variable was set to 0 and 100 respectively.

Note: *Fully qualified class names in the search template tells the matcher exactly which class to match for the static method call, but it will be matched with either the FQ name or the short name in the source code. Similarly, the fully qualified class name in the replace template is left as-is if the option 'Shorten fully qualified names' (Figure 2) is turned off. If it is turned on, then the short name of the class is used, possibly with an import statement added.*