AspectWerkz Pattern Language

About Terracotta Documentation

This documentation is about Terracotta DSO, an advanced distributed-computing technology aimed at meeting special clustering requirements.

Terracotta products without the overhead and complexity of DSO meet the needs of almost all use cases and clustering requirements. To learn how to migrate from Terracotta DSO to standard Terracotta products, see Migrating From Terracotta DSO. To find documentation on non-DSO (standard) Terracotta products, see Terracotta Documentation. Terracotta release information, such as release notes and platform compatibility, is found in Product Information.

Release: 3.6
Publish Date: November, 2011 **Documentation Archive »**

Unknown macro: {HTMLcomment}

examples using tc config-file elements would be useful

AspectWerkz Pattern Language

AspectWerkz supports a fine-grained pattern language for selecting join points. This is useful in Terracotta XML configuration elements in which a number of related abstractions must be selected. Instead of explicitly naming each member of the classes, methods, or fields required, an AspectWerkz pattern can be used in the configuration element to cover sets of related abstractions.

Wildcards

You can utilize two types of wildcards when constructing your patterns:

* - which is used as a regular wildcard. Matches for example only one package level or one method parameter. When used to match a package name, matches at least one character. Else match zero or more character.

`..` - used either to match any sequence of characters that start and end with a "." (so it can be used to pick out all types in any subpackage) or in method selectors to match as many parameters as possible.

For example `org.codehaus..*` will match all classes in all subpackages starting from `org.codehaus`.

`* method(..)` will match all methods with any number of parameters.
Note: you can only use the `..` wildcard as the "last" thing specified. I.e. this is not possible: `foo.bar..test.MyClass`, but this is: `foo.bar...` The same thing holds for method parameters.

Another specific character to express inheritance based matching will be presented further.

Combining the patterns

The patterns normally consists of a combination of a class and a method pattern or a class and a field pattern.

Example of a full method pattern:

```
<annotations> <modifiers> <return_type_pattern> <package_and_class_pattern>.<method_name_pattern>
(<parameter_type_patterns>)
```

Example of a full field pattern:

```
<annotations> <modifiers> <field_type_pattern> <package_and_class_pattern>.<field_name_pattern>
```

Example of a full class pattern:

```
<annotations> <modifiers> <package_and_class_pattern>
```

Class selections

The classes are selected by specifying a pattern that consists of:

- the annotations
- the modifiers
- the full name of the class

All class patterns must follow this structure:

```
<annotations> <modifiers> <full_class_name>
```

For the class selections specify the full package name of the class along with some wildcards.

Examples

`foo.bar.*`

- will match `foo.bar.FooBar2`
- as well as `foo.bar.FooBear`
- but not `foo.bar.subpackage.FooMouse`

`foo.*.FooBar`

- will match `foo.bar.FooBar`
- as well as `foo.bear.FooBar`
- but not `foo.bear.FooBear`

```
foo.*.FooB*
```

- will match `foo.bar.FooBar2`
- as well as `foo.bear.FooBear`
- as well as `foo.bear.FooB`

```
public foo.bar.*
```

- will match `public static final foo.bar.FooBar`
- as well as `public static foo.bar.FooBar`
- but not `static foo.bar.FooBar` or `private foo.bar.FooBar`

```
@Session foo.bar.*
```

- will match `@Session foo.bar.FooBar`
- but not `foo.bar.FooBar` or `private foo.bar.FooBar`

```
foo..
```

- will match all classes in all packages starting with foo

Method selections

The methods are selected by specifying a pattern that consists of:

- the annotations
- the modifiers
- the return type
- the full name of the method (including class and package)
- the parameter types

All method patterns must follow this structure:

<annotations> <modifiers> <return_type> <full_method_name>(<parameter_types>)

Examples

```
int foo.*.Bar.method()
```

- will match `int method()`
- but not `int method(int i)`

```
int *.method(*)
```

- will match `int Foo.method(int i)`
- but not `int Foo.method()`
- and not `int apackage.Foo.method(int i)`

```
* method(..)
```

- will match `void Foo.method()`
- as well as `void apackage.Bar.method(int[] i)`

```
int foo.*.*.method(*,int)
```

- will match `int method(String s, int i)`
- as well as `int method(int i1, int i2)`

```
int foo.*.Bar.method(..)
```

- will match `int method()`
- as well as `int method(String s, int i)`
- as well as `int method(int i, double d, String s, Object o)`

```
int foo.*.Bar.method(int,..)
```

- will match `int method(int)`
- as well as `int method(int i, String s)`
- as well as `int method(int i, double d, String s, Object o)`

```
int foo.*.Bar.method(java.lang.*)
```

- will match `int method(String s)`

- as well as `int method(StringBuffer sb)`

```
int foo.*.Bar.me*o*()
```

- will match `int method()`
- as well as `int metamorphosis()` and `int meo()`
- but not `int me()`

```
* foo.*.Bar.method()
```

- will match `int method()`
- as well as `java.lang.String method()`

```
java.lang.* foo.*.Bar.method()
```

- will match `java.lang.String Bar.method()`
- as well as `java.lang.StringBuffer Bar.method()`

```
static int foo.*.Bar.method()
```

- will match `static int method()`
- but not `int method(int i)`

```
@Transaction * foo.*.*.*(..)
```

- will match `@Transaction int method()`
- but not `void method(int i)`

Constructor selections

The constructors are selected by specifying a pattern that consists of:

- the annotations
- the modifiers
- the fully qualified name of the class (including package) plus the word 'new' as constructor name
- the parameter types

All the patterns must follow this structure:

```
<annotations> <modifiers> <className>.new(<parameter_types>)
```


Terracotta is not using constructor selectors and to match on constructors you should use method selector for `__INIT__` method instead. So instead of `Foo.new(..)` use `* Foo.__INIT__(..)`

Examples

```
foo.*.Bar.new()
```

- will match {{new Bar()
- but not `new Bar(int i)`

```
* new(..)
```

- will match `new Foo()`
- as well as `new apackage.Bar(int[] i)`

```
*.new(String)
```

- will match `new Foo(String name)` and `new Bar(String name)`
- but not `new Foo()`

Field selections

Fields are selected by specifying a pattern that consists of:

- the annotations
- the modifiers
- the field type
- the full name of the field (including class and package)

All field patterns must follow this structure:

```
<annotations> <modifiers> <field_type> <full_field_name>
```

Note that Terracotta currently doesn't support selection of excluded or transient fields based on annotations.

Examples

`int foo.*.Bar.m_foo`

- will match `int m_foo`
- but not `int s_foo` or `long m_foo`

`* m_field`

- will match `int Foo.m_field`
- as well as `int[] apackage.Bar.field`

`* foo.*.Bar.m_foo`

- will match `int m_foo`
- as well as `java.lang.String m_foo`

`java.lang.* foo.*.Bar.m_foo`

- will match `java.lang.String m_foo`
- as well as `java.lang.StringBuffer m_foo`

`int foo.*.Bar.m_*`

- will match `int m_foo`
- as well as `int m_bar`

`int foo.*.Bar.m_*oo*`

- will match `int m_foo`
- as well as `int m_looser`
- as well as `int m_oo`

Subtype patterns

It is possible to pick out all subtypes of a type with the "+" wildcard. The "+" wildcard immediately follows a type name pattern.

The following picks out all method-call join points where an instance of exactly type `Bar` is constructed:

`* foo.Bar.*(..)`

The following picks out all method-call join points where an instance of *any subtype of* `Bar` (including `Bar` itself) is constructed:

`* foo.Bar+.*(..)`

Note that `foo.Bar` can be a class (including a super class) or an interface.

Array type patterns

A type name pattern or subtype pattern can be followed by one or more sets of square brackets to make array type patterns. So `java.lang.Object[]` is an array type pattern, and so is `foo.bar.*[][]`.

Abbreviations

When picking out the return and parameter types it is possible to use predefined abbreviations for the classes in the `java.lang.*` and `java.util.*` packages. If you specify only the class name it will be mapped to the full class name for the class (you cannot use patterns in abbreviations).

Abbreviations are supported for array types as well, with a dimension less or equal to 2. `String[][]` will thus be resolved as `java.lang.String[][]` but `String[][][]` will not.

This is usefull when dealing with complex advice signature in the XML definition.

Examples

You can use:

- String instead of `java.lang.String`
- List instead of `java.util.List`

- but not `String*` instead of `java.lang.String` or `java.lang.StringBuffer`
- and so on...

Apart from these abbreviations you always have to specify the fully qualified name of the class (along with the wildcards).