SmartSpace®

Refactoring Support in SmartSpace

From version 3.6

Ubisense Limited, St Andrew's House, St Andrew's Road, Cambridge CB4 1DL, United Kingdom.

Telephone: +44 (0)1223 535170. Website: https://www.ubisense.com

Ubisense Limited
St Andrew's House
St Andrew's Road
Cambridge CB4 1DL
United Kingdom

Tel: +44 (0)1223 535170

WWW: *https://www.ubisense.com*

Contents

Refactoring Support in SmartSpace

The Ubisense refactoring tool provides a means to successfully replace type or property definitions in your dataset.

The following describes the refactoring tool and how it can be used.

Introduction and motivation

Building a data model involves some related activities that proceed roughly in sequence like this:

1. Describing the kinds of thing in the model by using types and an inheritance hierarchy
2. Describing the properties of those types and the relationships between them using simple and complex properties
3. Describing the relationships between properties by using rules and event handlers
4. Describing how end users will interact with the model by using searches and web forms

Every model is a sequence of decisions, and not every decision is easy to make at the time. When building a model it is easy to make a poor decision in an early stage without finding out until much later when you've already done a lot of work that depends on all the decisions you've made up to now (including the poor ones). The goal of our refactoring support is to enable you to fix up poor design decisions without having to tear up everything that depends on them.

The most common poor design decisions include:

- Getting the name of a type or property wrong
- Putting the arguments of a complex property in the wrong order
- Putting a type in the wrong position in an inheritance hierarchy

The Ubisense refactoring tool allows you to fix all these problems by replacing a type or property definition with another definition that has the correct name, argument order, or position in the inheritance hierarchy. The refactoring tool takes care of propagating the replacement throughout the data model so that rules, event handlers, searches and reports are all updated accordingly.

Experienced users will know that it is possible to some extent to fix up the first two problems above by using the localization support to translate type and property names and even reorder property arguments in order to present the underlying definition differently to different end users. But refactoring support is a better approach in cases where the data model is still under development, where the underlying argument order might matter for performance reasons, or where the underlying property is going to be used by external programmers (e.g. via the .NET or RESTful APIs).

Overview of the refactoring tool

Licensing

The refactoring tool is licensed as part of the Rules engine developer.

To get the tool, run **Application Manager**, open the **DOWNLOADABLES** task, and expand **SmartSpace core/Refactoring support**. Select **ubisense_refactor.exe** and click **Download selected items**. Optionally, change the download destination directory, and then click **Start download**.

Executable program

The refactoring tool is a command-line tool, separate from SmartSpace Config or **ubisense_udm_admin.exe**. This is because refactoring works on the untranslated underlying names of the definitions, while SmartSpace Config or **ubisense_udm_admin.exe** work on translated names; because translation can itself change the names types and properties, and change the argument orders of properties, using refactoring on translated names would create too many possibilities for confusion.

The refactoring tool **ubisense_refactor.exe** has this usage message:

```
$ ./ubisense_refactor.exe
usage: ubisense_refactor.exe --type [from to]
     | ubisense_refactor.exe --property [from to Int*]
     | ubisense_refactor.exe --version
     | ubisense_refactor.exe --help
--type     If from, to are not defined then print the list of all available types
           or, if they are defined, replace 'from' with 'to' across the dataset.
--property If from, to are not defined then print the list of all available properties
           or, if they are defined, replace 'from' with 'to' across the dataset.
           using an optional argument mapping as a sequence of numbers used to
           reorder the arguments of 'from'.  For example, if from is P and to is Q
           then 'refactor --property P Q 2 1 0' will replace instances of P(x,y,z)
           by Q(z,y,x).
--version  Print version information
--help     Print this help information
```

Examples of tool use

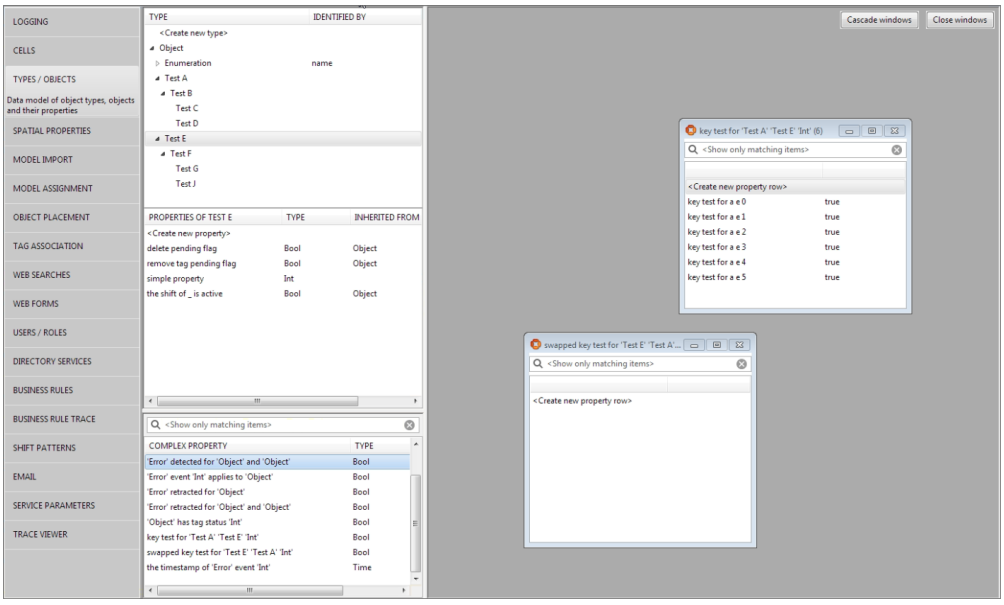
Property replacement

In this example, we have two types, ‘Test A’ and ‘Test E’ and two properties, ‘key test for ‘Test A’ ‘Test E’ Int : Bool’ and ‘swapped key test for ‘Test E’ ‘Test A’ Int : Bool’.

There are some rules and event handlers, searches etc. defined for the ‘key test’ property, and none defined for the ‘swapped key test’ property. In one event handler for the key test property we have these instructions:

```
when key test for test a test e int becomes true do
    if int > 0 then
        set key test for test a test e int - 1 to true
```

so that inserting the single value ‘key test for “a” “e” 5’ results in a total of six rows as shown below:



If we run the refactoring tool we can see the various internal names corresponding to these types and properties:

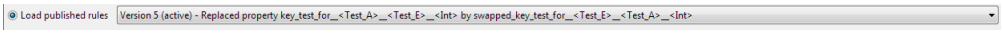
```
$ ./ubisense_refactor.exe --type
[Custom]Logic_Error
[Custom]Sensing_Error
[Custom]Process_Error
Object
[Custom]Error
[Custom]Enumeration
[Custom]Integration_Error
Test_B
Test_E
Test_A
Test_G
Test_C
Test_J
Test_F
Test_D

$ ./ubisense_refactor.exe --property
the_shift_of__is_active
[Custom]name<[Custom]Enumeration>
[Custom]sequence_number<[Custom]Error>
key_test_for__<Test_A>__<Test_E>__<Int>
[Custom]__<[Custom]Error>retracted_for__<Object>
[Custom]the_timestamp_of__<[Custom]Error>event__<Int>
swapped_key_test_for__<Test_E>__<Test_A>__<Int>
[SmartSpaceCore]__<Object>has_tag_status__<Int>
[SmartSpaceCore]remove_tag_pending_flag<Object>
[SmartSpaceCore]delete_pending_flag<Object>
[Custom]lifetime_in_hours<[Custom]Error>
[Custom]__<[Custom]Error>detected_for__<Object>
simple_property<Test_A>
simple_timeout<Test_A>
simple_property<Test_E>
[Custom]__<[Custom]Error>retracted_for__<Object>and__<Object>
[Custom]__<[Custom]Error>detected_for__<Object>and__<Object>
[Custom]__<[Custom]Error>event__<Int>applies_to__<Object>
```

Now we want to replace all occurrences of ‘key test for ‘Test A’ ‘Test E’ Int : Bool’ with occurrences of ‘swapped key test for ‘Test E’ ‘Test A’ Int : Bool’ across the dataset. As well as changing the names everywhere, this also involves changing the order in which arguments instantiated: the initial two arguments of the old property need to be swapped round when used in the new property.

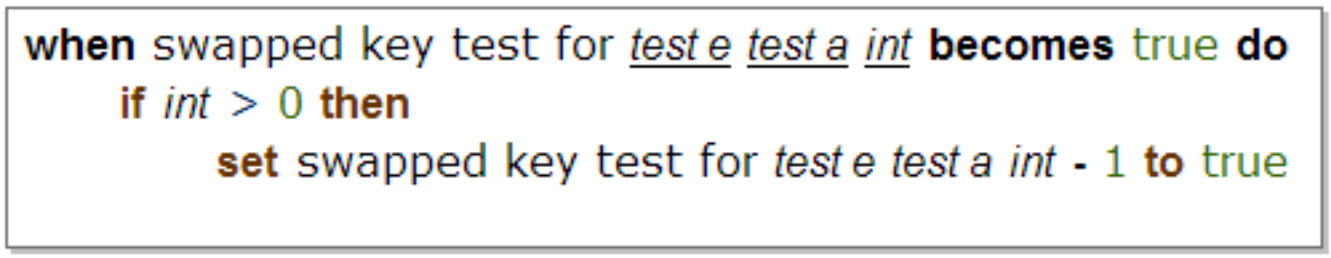
To do this, we use this command in the refactoring tool:

```
$ ./ubisense_refactor.exe --property 'key_test_for__<Test_A>__<Test_E>__<Int>'
'swapped_key_test_for__<Test_E>__<Test_A>__<Int>' 1 0 2
replacing 'key_test_for__<Test_A>__<Test_E>__<Int>' with 'swapped_key_test_for__<Test_
E>__<Test_A>__<Int>'
performing rule substitutions
performing substitutions in web search config
performing substitutions in web form config
```

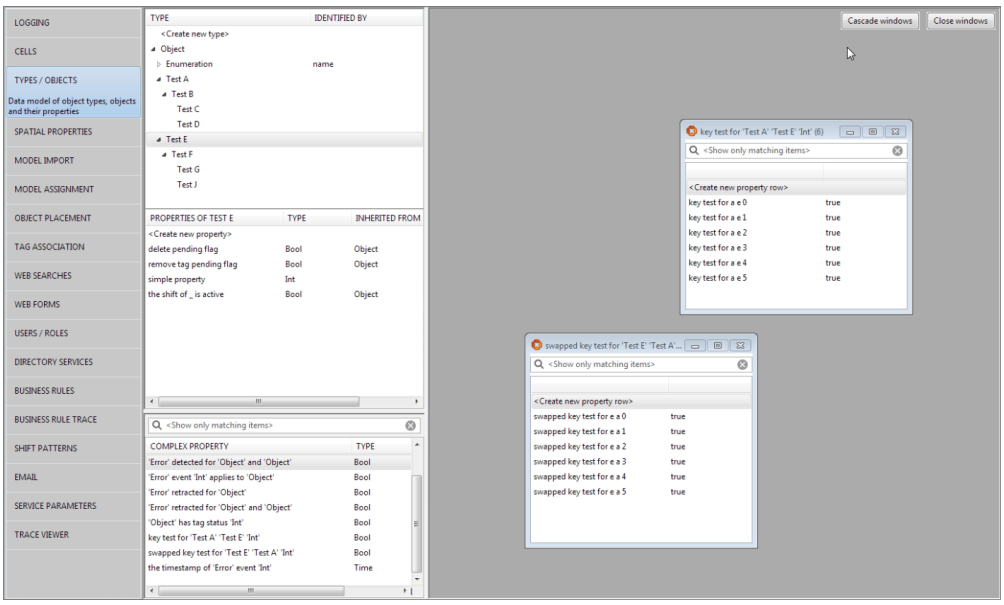
Now all occurrences of the old property are replaced by the new property. For example, if we look at the rules configuration, we find a new set of published rules available to be loaded:



and if we load them, we find that our example event handler has been rewritten to be this, with the property name changed and its arguments reordered appropriately:



the rows of ‘key test’ have been copied to ‘swapped key test’:



and all the other references across the dataset have been changed, replacing the old property with the new one.

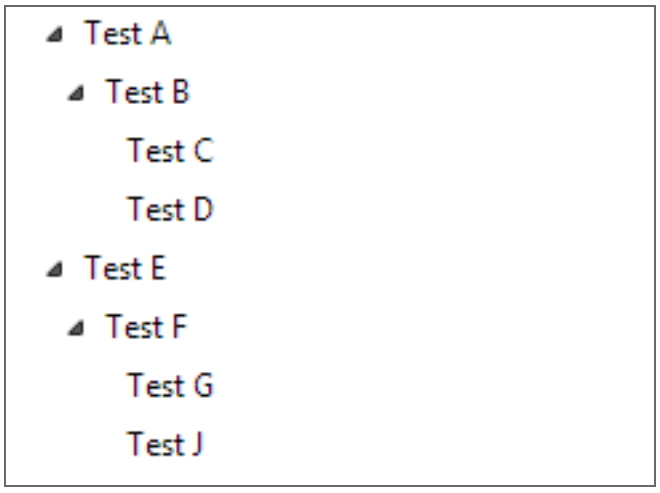
Note that the old property and its rows are still retained, but at this point it will be possible to delete old property's rows and then delete the property itself, because all references to the old property will have been replaced by references to the new one. Also note that, because the old property is still there after the replacement, it is possible to undo the replacement of 'old' by 'new' by doing another replacement of 'new' by 'old'.

For more details of exactly what the property replacement does, and the order in which it does it, see *Property replacement in detail*.

Type replacement

Type replacement is a little more complicated than property replacement and is not reversible in the way that property replacement is reversible. Put simply, when a type T is replaced by a type U, all the properties and descendants of T are deleted and recreated as properties and descendants of U, and then T itself is deleted. Because type replacement involves deleting the old type, the old type must be empty for the replacement to be permitted; this is because we do not want a situation where there are objects in the dataset whose type does not exist.

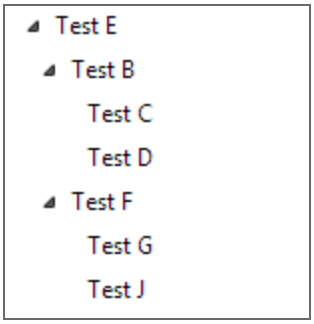
In the previous example of property replacement, we had a type hierarchy looking like this:



and there were various properties of Test A (including the 'old' property in the property replacement). If we now execute the command to replace Test A with Test E, we get this output:

```
$ ./client/i586_windows_1.3_dyn/ubisense_refactor.exe --type Test_A Test_E
replacing 'Test_A' with 'Test_E'
checking validity of the type replacement operation
performing type substitutions in tag constraints
UserDataModel::Test_A descendant count = 3
performing type replacement in user data model
delete property key_test_for__<Test_A>__<Test_E>__<Int>
delete property swapped_key_test_for__<Test_E>__<Test_A>__<Int>
delete property simple_property<Test_A>
delete property simple_timeout<Test_A>
delete type Test_C
delete type Test_D
delete type Test_B
delete type Test_A
create type Test_B with parents [Test_E]
retrying due to transient error: '*** Test_B ***' is already defined as a type
retrying due to transient error: '*** Test_B ***' is already defined as a type
create type Test_D with parents [Test_B]
create type Test_C with parents [Test_B]
modify all translations of key_test_for__<Test_A>__<Test_E>__<Int> because it has been
renamed to key_test_for__<Test_E>__<Test_E>__<Int>
create complex property key_test_for__<Test_E>__<Test_E>__<Int>: [Test_E,Test_E,Int]-
>Bool
set all property translations for key_test_for__<Test_A>__<Test_E>__<Int>
modify all translations of swapped_key_test_for__<Test_E>__<Test_A>__<Int> because it
has been renamed to swapped_key_test_for__<Test_E>__<Test_E>__<Int>
create complex property swapped_key_test_for__<Test_E>__<Test_E>__<Int>: [Test_E,Test_
E,Int]->Bool
set all property translations for swapped_key_test_for__<Test_E>__<Test_A>__<Int>
modify all translations of simple_property<Test_A> because it has been renamed to
simple_property<Test_E>
create simple property simple_property<Test_E>: Test_E->Int
set all property translations for simple_property<Test_A>
modify all translations of simple_timeout<Test_A> because it has been renamed to
simple_timeout<Test_E>
create simple property simple_timeout<Test_E>: Test_E->Time
set all property translations for simple_timeout<Test_A>
performing rule substitutions
performing substitutions in web search config
performing substitutions in web form config
```

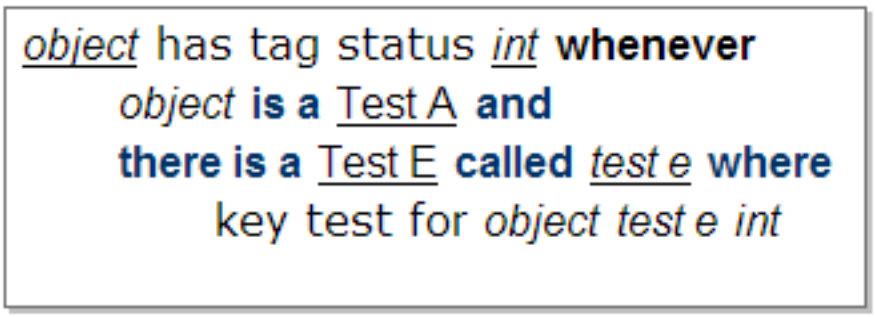
Now if we look at SmartSpace Config we see quite a few changes. First, the inheritance hierarchy is changed:



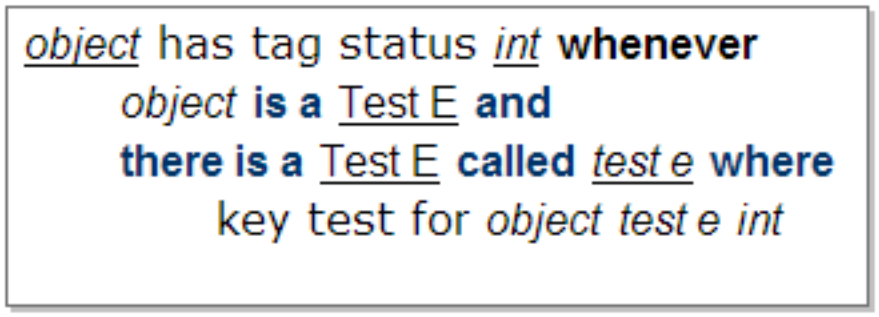
the properties involving Test A have disappeared, to be replaced by:

key test for 'Test E' 'Test E' 'Int'	Bool
swapped key test for 'Test E' 'Test E' 'Int'	Bool

Looking at the latest rule definitions we see that in rules mentioning Test A and ‘key test for ‘Test A’ ‘Test E’ Int : Bool’, these have been replaced by Test E and a new property ‘key test for ‘Test E’ ‘Test E’ Int : Bool’, i.e. turning this definition:



into this:



Similar changes have been made across the entire dataset. Test A is no more, and wherever it occurred, Test E has been introduced in its place.

For more details of exactly what the type replacement does, and the order in which it does it, see *Type replacement in detail*.

Refactoring tool operation in detail

The refactoring tool operates on the underlying untranslated types and properties and allows the user to replace one type with another (already defined) type, or to replace one property with another (already defined) property and an optional reordering of the arguments.

To preserve the correctness of the dataset, the replacement types and properties must obey certain constraints, so that every reference to the old definition in the dataset can legally be replaced by a suitably modified reference to the new definition.

Property replacement in detail

How a property replacement is specified in general

Given two properties:

- P with a sequence of argument types A_0, … , A_n and result type R, and
- Q with a sequence of argument types B_0, … , B_n and result type S

then we can describe a replacement of P by Q using the triple (P, Q, M). In this triple, M is a list of integers [M_0, … , M_n]), where each of M_0, … , M_n is a number from 0 to n such that B_i (the argument number i of Q) corresponds to A_{Mi} (the argument number M_i of P).

For example, the replacement

(<Person>is a parent of<Person> : Bool , <Person>is a child of<Person> : Bool , [1,0])

would replace all instances of “Daedalus is a parent of Icarus” with “Icarus is a child of Daedalus”.

The simple case of property replacement without argument reordering

In the refactoring tool, the argument mapping can be left out, and then it is assumed that there will be no property reordering (which is equivalent to an argument mapping of [0,1,2,…,n]).

Constraints on property replacements

To ensure correctness across the dataset, a property replacement (P, Q, M) is only permitted if:

1. *P* and *Q* both exist
2. The result types of *P* and *Q* are identical
3. The arities (number of arguments) of *P* and *Q* are identical

4. If the arity of *P* (and *Q*) is *n*, for each *i* where $0 \le i < n$, the type of the *i* th argument of *Q* must be identical to the type of the M_i th argument of *P*.

The effect of a property replacement

When a property replacement (P,Q,M) is completed the following data is changed:

1. All rows of P are copied to rows of Q, if and only if P and Q are site level properties
2. If P is a spatial property, then all instances of P are copied to be instances of Q
3. The currently-active set of rules is loaded, all references to P are replaced with references to Q, and the resulting set of rules is published, creating a new active set of rules
4. If the Visibility component is licensed, then all references to P are replaced with references to Q in the specifications of web searches and web forms

Data preserved by a property replacement

When a property replacement (P,Q,M) is completed, neither the property P nor its contents are deleted. However, there should remain no references to P, so it is safe to delete P and its contents after the replacement is completed.

Type replacement in detail

How a type replacement is specified

Given two types T and U, a type replacement is completely specified by the pair (T,U).

Constraints on type replacements

To ensure correctness across the dataset, a type replacement of T by U is only permitted if:

1. Both T and U exist
2. The type T is a user-defined type
3. U is not a descendant of T (but it is fine for T to have other descendants, which will end up as descendants of U instead)
4. There are no objects of type T in the user data store or naming schema (i.e. T and any descendant types of T must be empty)
5. Moving descendant types of T to be descendant types of U does not create a situation where one of these types has two name properties

6. All properties inherited by T must also be inherited by U (i.e. it is OK for T and U to have different parents as long as T does not thereby have properties that U does not have)
7. Every property P that explicitly mentions T in its argument or result type must be transformable into a property P′ (which is exactly like P′ but with all mentions of T replaced by mentions of U) without clashing with some preexisting property of U (e.g. if T has a simple property p of type Int then it is OK for U to have a preexisting simple property p of type Int, but it is not OK for U to have a preexisting simple property p of type Bool)

The effect of a type replacement

When a type replacement (T,U) is completed the following data is changed:

1. Every property that mentions T is deleted, but replaced with a similar property in which all mentions of T have been replaced by U, and all translations of the deleted property are transferred over to the new one
2. All descendant types of T are moved to be descendant types of U (this means that each of the types is deleted together with all its properties, and then recreated as a descendant of U together with all its properties)
3. All property replacements resulting from (1) are carried out as described in the section on property replacements (note that these replacements always have a default argument map and are always legal, and since the property T must be empty there are no rows to copy)
4. All references to T are replaced by references to U in the platform support for tag constraints and representation model ownership
5. All references to T are replaced by references to U in the SmartSpace core support for spatial property ownership
6. The currently-active set of rules is loaded, all references to T are replaced with references to U and the resulting set of rules is published, creating a new active set of rules. This change is combined with the change described in (2) so that there is one new published set of rules that results from the simultaneous application of the replacement of T by U and all the property replacements that this implies
7. If the Visibility component is licensed, then all references to T are replaced with references to U in the specifications of web searches and web forms