Creatio Academy

Back-end development

Replacing class factory

Version 8.0

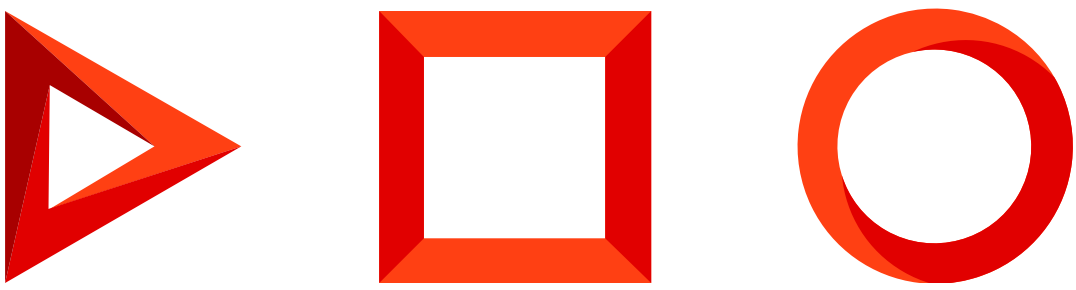


Table of Contents

Replacing class factory

Advanced

Instantiate the replacing class using the **replacing class object factory**. Creatio requests the replaced type instance from the factory. The factory returns the instance of the corresponding replacing type, which the factory computes using the dependency tree in source code schemas.

To **create a replacing configuration element** of the appropriate type, follow the procedure covered in a separate article: Develop configuration elements.

[Override] attribute

The `[Override]` attribute type belongs to the `Terrasoft.Core.Factories` namespace and directly inherits the `System.Attribute` base type. Learn more about the `Terrasoft.Core.Factories` namespace in the .NET class library. Learn more about the `System.Attribute` base type in the official Microsoft documentation.

The `[Override]` attribute only applies to classes. The **purpose** of the `[Override]` attribute is to define classes to take into account when building a dependency tree of replacing and replaced factory types.

View how to apply the `[Override]` attribute to the `MySubstituteClass` replacing class below. `SubstitutableClass` is the replaced class.

Template to apply the [Override] attribute

```
[Override]
public class ReplacingClassName : ReplacedClassName
{
    /* Implement the class. */
}
```

Example that applies the [Override] attribute

```
[Override]
public class MySubstituteClass : SubstitutableClass
{
    /* Implement the class. */
}
```

ClassFactory class

The **purpose** of the `ClassFactory` class is to implement the factory of replacing Creatio objects. The factory uses

the Ninject open-source framework to inject dependencies. Learn more about the `ClassFactory` static class in the .NET class library.

Creatio **initializes the factory** during the first call to it, i. e. at the first attempt to retrieve the replacing type instance. The factory collects data about the replaced configuration types as part of the initialization.

The **operating procedure of the factory**:

1. Search for replacing types. The factory analyzes the configuration build types. The factory interprets the class marked with the `[Override]` attribute as a replacing class and its parent class as a replaced class.
2. Build the type dependency tree as a list of `Replaced type → Replacing type` value pairs. The replacement hierarchy tree does not take transitional types into account.
3. Replace the source class with the last inheritor in the replacement hierarchy.
4. Bind replacement types based on the type dependency tree. The factory uses the Ninject framework.

View the factory workflow based on the class hierarchy example below.

Class hierarchy example

```
/* Source class. */
public class ClassA { }

/* Class that replaces ClassA. */
[Override]
public class ClassB : ClassA { }

/* Class that replaces ClassB. */
[Override]
public class ClassC : ClassB { }
```

The factory will use the class hierarchy to build a dependency tree you can view below.

Dependency tree example

```
ClassA → ClassC
ClassB → ClassC
```

The factory will not take the transitional types into account when building the dependency tree.

Type replacement hierarchy example

```
ClassA → ClassB → ClassC
```

Here, the `ClassC` type, and not the transitional `ClassB` type, replaces `ClassA`. This is because `ClassC` is the last

inheritor in the replacement hierarchy. As such, the factory will return the `ClassC` instance if you request the `ClassA` or `ClassB` type instance.

Replaced type instance

To **retrieve a replaced type instance**, use the `Get<T>` public static parameterized method. The `ClassFactory` factory provides this method. The replaced type serves as a generic method parameter. Learn more about the `Get<T>` method in the .NET class library.

Example that retrieves the replaced type instance

```
var substituteObject = ClassFactory.Get<SubstitutableClass>();
```

As a result, Creatio will create the `MySubstituteClass` class instance. You do not have to state the type of the new instance explicitly. The preliminary initialization lets the factory determine the type to replace the requested type and create the corresponding instance.

The `Get<T>` method can accept an array of `ConstructorArgument` objects as parameters. Each of the objects in the array is a class constructor argument the factory created. As such, the factory lets you instantiate replacing objects with parameterized constructors. The factory core resolves the dependencies required for the creation or operation of the object independently.

We recommend ensuring the signature of the replaced class constructors matches the signature of the replacing class. If the replacing class implementation logic must declare the constructor that has a custom signature, make sure to follow the rules listed below.

Rules for creating and calling replaced and replacing class constructors:

- If the replaced class **lacks an explicitly parameterized constructor** (the class only has a default constructor), you can implement an explicitly parameterized independent constructor in the replacing class without any restrictions. Follow the standard order of calls to the parent (replaced) and child (replacing) class constructors. When instantiating the replaced class via the factory, make sure to pass the correct parameters to initialize the replacing class properties to the factory.
- If the replaced class **has a parameterized constructor**, implement the constructor in the replacing class. The replacing class constructor must call the parameterized constructor of the parent (replaced) class explicitly and pass the parameters for the correct initialization of the parent properties to the parent class. Other than that, the replacing class constructor may initialize its properties or remain empty.

Failure to comply with the rules will result in a runtime error. The developer is responsible for the correct initialization of replacing and replaced class properties.

Replacing class management examples

Example 1

The `SubstitutableClass` replaced class has one default constructor.

`SubstitutableClass` replaced class

```
/* Declare a replaced class. */
public class SubstitutableClass
{
    /* The class property to initialize in the constructor. */
    public int OriginalValue { get; private set; }

    /* The default constructor that initializes the OriginalValue property with 10. */
    public SubstitutableClass()
    {
        OriginalValue = 10;
    }

    /* The method that returns the OriginalValue multiplied by 2. You can redefine this method i
    public virtual int GetMultipliedValue()
    {
        return OriginalValue * 2;
    }
}
```

Declare two constructors in the `SubstituteClass` replacing class: the default constructor and the parameterized constructor. Redefine the `GetMultipliedValue()` parent method using the replacing class.

`SubstituteClass` replacing class

```
/* Declare a class to replace the SubstitutableClass. */
[Terrasoft.Core.Factories.Override]
public class SubstituteClass : SubstitutableClass
{
    /* The SubstituteClass class property. */
    public int AdditionalValue { get; private set; }

    /* The default constructor that initializes the AdditionalValue property with 15. You do not
    public SubstituteClass()
    {
        AdditionalValue = 15;
    }

    /* The parameterized constructor that initializes the AdditionalValue property to the value
    public SubstituteClass(int paramValue)
    {
        AdditionalValue = paramValue;
    }

    /* Replace the parent method. The method returns the AdditionalValue multiplied by 3. */
```

```
    public override int GetMultipliedValue()
    {
        return AdditionalValue * 3;
    }
}
```

Example. Instantiate and call the `GetMultipliedValue()` method of the `SubstituteClass` replacing class using the default constructor and the parameterized constructor.

See the examples of retrieving the `SubstituteClass` replacing class instance using the factory below.

Examples of retrieving the replacing class using the factory

```
/* Retrieve an instance of the class to replace the SubstitutableClass. The factory returns a Su
var substituteObject = ClassFactory.Get<SubstitutableClass>();

/* The variable equals 10. The default parent constructor initializes the OriginalValue property
var originalValue = substituteObject.OriginalValue;

/* Call the replacing class method that returns the AdditionalValue multiplied by 3. The variabl
var additionalValue = substituteObject.GetMultipliedValue();

/* Retrieve the replacing class instance initialized by the parameterized constructor. The Const
var substituteObjectWithParameters = ClassFactory.Get<SubstitutableClass>(
    new ConstructorArgument("paramValue", 20));

/* The variable equals 10. */
var originalValueParametrized = substituteObjectWithParameters.OriginalValue;

/* The variable equals 60 since the AdditionalValue was initialized with 20. */
var additionalValueParametrized = substituteObjectWithParameters.GetMultipliedValue();
```

Example 2

The `SubstitutableClass` replaced class has 1 parameterized constructor.

`SubstitutableClass` replaced class

```
/* Declare a replaced class. */
public class SubstitutableClass
{
    /* The class property to initialize in the constructor. */
    public int OriginalValue { get; private set; }
```

```
    /* The parameterized constructor that initializes the OriginalValue property to the value pa
    public SubstitutableClass(int originalParamValue)
    {
        OriginalValue = originalParamValue;
    }

    /* The method that returns the OriginalValue multiplied by 2. You can redefine the method ir
    public virtual int GetMultipliedValue()
    {
        return OriginalValue * 2;
    }
}
```

The `SubstituteClass` replacing class has 1 parameterized constructor as well.

`SubstituteClass` replacing class

```
/* Declare the class to replace the SubstitutableClass. */
[Terrasoft.Core.Factories.Override]
public class SubstituteClass : SubstitutableClass
{
    /* The SubstituteClass class property. */
    public int AdditionalValue { get; private set; }

    /* The parameterized constructor that initializes the AdditionalValue property to the value
    public SubstituteClass(int paramValue) : base(paramValue + 8)
    {
        AdditionalValue = paramValue;
    }

    /* Replace the parent method. The method returns AdditionalValue multiplied by 3. */
    public override int GetMultipliedValue()
    {
        return AdditionalValue * 3;
    }
}
```

Example. Create and use an instance of the `SubstituteClass` replacing class.

See the example of creating and using an instance of the `SubstituteClass` replacing class using the factory below.

Example of creating and using a replacing class instance using the factory

```
/* Retrieve an instance of the replacing class initialized in the parameterized constructor. The
var substituteObjectWithParameters = ClassFactory.Get<SubstitutableClass>(
    new ConstructorArgument("paramValue", 10));

/* The variable equals 18. */
var originalValueParametrized = substituteObjectWithParameters.OriginalValue;

/* The variable equals 30. */
var additionalValueParametrized = substituteObjectWithParameters.GetMultipliedValue();
```

Example 3

The `SubstitutableClass` replaced class has 1 parameterized constructor.

`SubstitutableClass` replaced class

```
/* Declare a replaced class. */
public class SubstitutableClass
{
    /* The class property to initialize in the constructor. */
    public int OriginalValue { get; private set; }

    /* The parameterized constructor that initializes the OriginalValue property to the value pa
    public SubstitutableClass(int originalParamValue)
    {
        OriginalValue = originalParamValue;
    }

    /* The method that returns the OriginalValue multiplied by 2. You can redefine this method i
    public virtual int GetMultipliedValue()
    {
        return OriginalValue * 2;
    }
}
```

Example. Redefine the `GetMultipliedValue()` parent method in the `SubstituteClass` replacing class.

The `SubstituteClass` replacing class redefines the `GetMultipliedValue()` method that will return a fixed value. You do not have to initialize the `SubstituteClass` class properties. However, the class requires an explicit declaration of the constructor that calls the parameterized parent constructor to initialize the parent properties correctly.

`SubstituteClass` replacing class

```
// Declare a class to replace SubstitutableClass.
[Terrasoft.Core.Factories.Override]
public class SubstituteClass : SubstitutableClass
{
    /* Empty default constructor that explicitly calls the parent class constructor to initializ
    public SubstituteClass() : base(0)
    {
    }

    /* You can also use an empty parameterized constructor to pass the parameters to the parent
    public SubstituteClass(int someValue) : base(someValue)
    {
    }

    /* Replace the parent method. The method will return a fixed value. */
    public override int GetMultipliedValue()
    {
        return 111;
    }
}
```

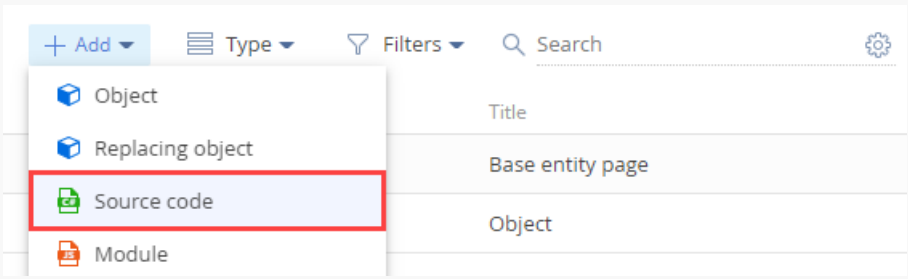
Create a replacing class

Advanced

Example. Create a replaced class and a custom web service that uses cookie-based authentication in a custom package. Create a replacement class in another custom package. Call the custom web service both using and not using the class replacement.

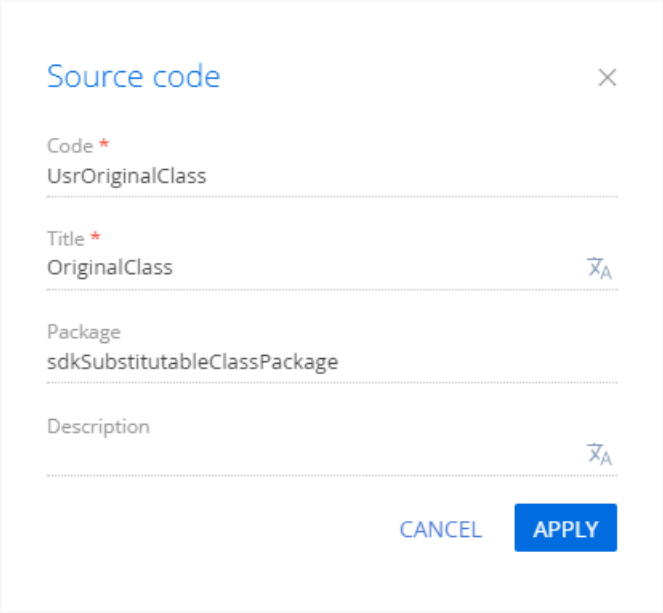
1. Implement the replaced class

1. Go to the [*Configuration*] section and select a custom package to which to add the schema.
2. Click [*Add*] → [*Source code*] on the section list toolbar.



3. Fill out the schema properties in the Schema Designer:

- [*Code*] – "UsrOriginalClass".
- [*Title*] – "OriginalClass".



4. Create a replaced `UsrOriginalClass` class that contains the `GetAmount(int, int)` virtual method. The method adds two values passed as parameters.

UsrOriginalClass

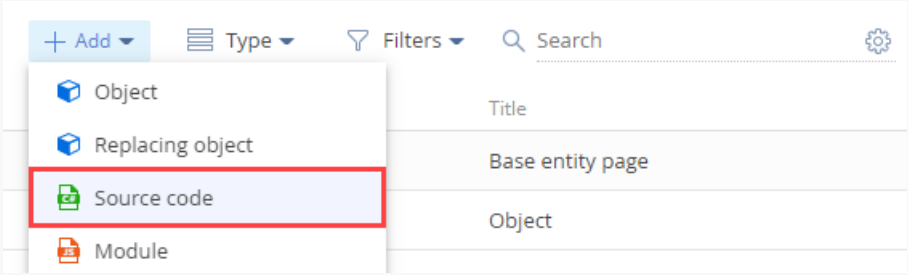
```
namespace Terrasoft.Configuration
{
    public class UsrOriginalClass
    {
        /* GetAmount() is a virtual method that has its own implementation. Inheritors can re
        public virtual int GetAmount(int originalValue1, int originalValue2)
        {
            return originalValue1 + originalValue2;
        }
    }
}
```

5. Click [*Save*] then [*Publish*] on the Designer's toolbar.

2. Implement a replacing class

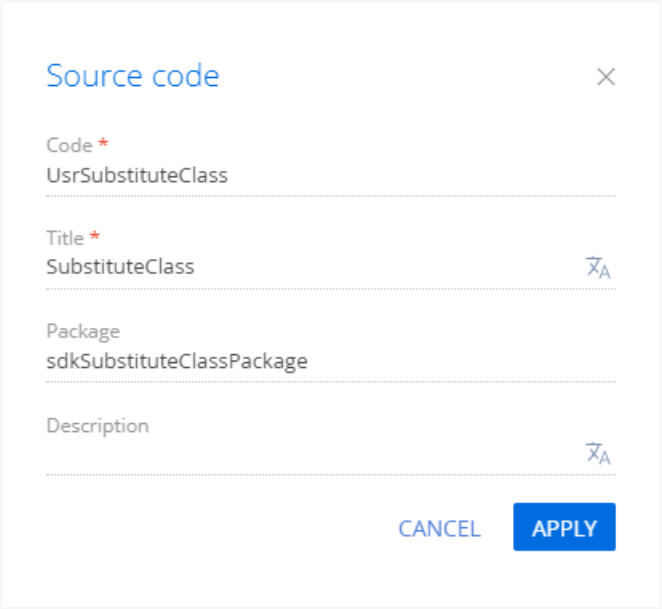
1. Go to the [*Configuration*] section and select a custom package to which to add the schema. Select a package other than the package in which you implemented the `UsrOriginalClass` replaced class.
2. Add the custom package that contains the `UsrOriginalClass` replaced class as a dependency for the custom package that contains the replacing class.

3. Click [*Add*] → [*Source code*] on the section list toolbar.



4. Fill out the schema properties in the Schema Designer:

   - [*Code*] – "`UsrSubstituteClass`".
   - [*Title*] – "`SubstituteClass`".



5. Create a `SubstituteClass` replacing class that contains the `GetAmount(int, int)` method. The method summarizes two values passed as parameters and multiplies the sum by the value passed in the `Rate` property. Creatio will initialize the `Rate` property in the replacing class constructor.

`UsrSubstituteClass`

```
namespace Terrasoft.Configuration
{
    [Terrasoft.Core.Factories.Override]
    public class UsrSubstituteClass : UsrOriginalClass
    {
        /* Rate. Assign the property value in the class. */
        public int Rate { get; private set; }

        /* Initialize the Rate property in the constructor using the passed value. */
```

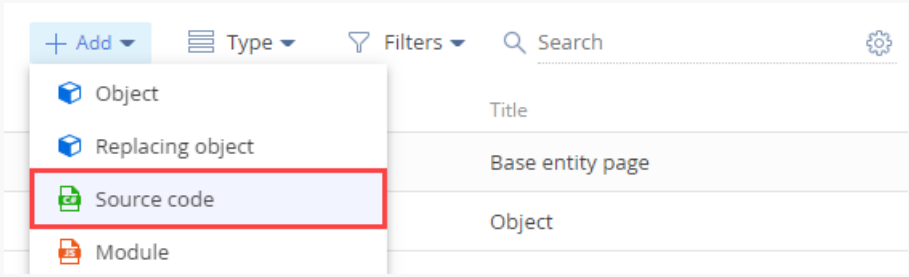
```
        public UsrSubstituteClass(int rateValue)
        {
            Rate = rateValue;
        }

        /* Replace the parent method using the custom implementation. */
        public override int GetAmount(int substValue1, int substValue2)
        {
            return (substValue1 + substValue2) * Rate;
        }
    }
}
```

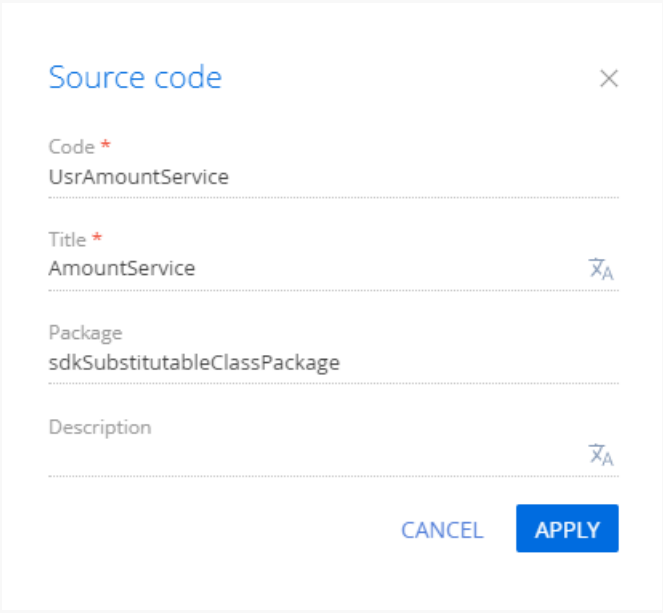
6. Click [*Save*] then [*Publish*] on the Designer's toolbar.

3. Implement a custom web service

1. Go to the [*Configuration*] section and select a custom package to which to add the schema. Use the package in which you implemented the `UsrOriginalClass` replaced class for the custom web service.
2. Click [*Add*] → [*Source code*] on the section list toolbar.



3. Fill out the schema properties in the Schema Designer:
   - [*Code*] – "`UsrAmountService`".
   - [*Title*] – "`AmountService`".



4. Create a service class.

   a. Add the `Terrasoft.Configuration` namespace in the Schema Designer.

   b. Add the namespaces the data types of which to utilize in the class using the `using` directive.

   c. Add the name of the `UsrAmountService` class that matches the schema name (the [*Code*] property).

   d. Specify the `Terrasoft.Nui.ServiceModel.WebService.BaseService` class as a parent class.

   e. Add the `[ServiceContract]` and `[AspNetCompatibilityRequirements(RequirementsMode = AspNetCompatibilityRequirementsMode.Required)]` attributes to the class.

5. Implement a class method.

   Add the `public string GetAmount(int value1, int value2)` method that implements the endpoint of the custom web service to the class in the Schema Designer. The `GetAmount(int, int)` method adds two values passed as parameters.

   View the source code of the `UsrAmountService` custom web service below.

UsrAmountService

```
namespace Terrasoft.Configuration
{
    using System.ServiceModel;
    using System.ServiceModel.Activation;
    using System.ServiceModel.Web;
    using Terrasoft.Core;
    using Terrasoft.Web.Common;

    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode = AspNetCompatibilityRequirementsMode.R
    public class UsrAmountService : BaseService
```

```
    {

        [OperationContract]
        [WebGet(RequestFormat = WebMessageFormat.Json, BodyStyle = WebMessageBodyStyle.Wrappe
        public string GetAmount(int value1, int value2) {
            /*
            // Create a source class instance using the class factory.
            var originalObject = Terrasoft.Core.Factories.ClassFactory.Get<UsrOriginalClass>(

            // Retrieve the GetAmount() method output. Pass the values of the page input fiel
            int result = originalObject.GetAmount(value1, value2);

            // Display the results on the page.
            return string.Format("The result value, retrieved after calling the replacement c
            */

            /*
            // Create a replacing class instance using the replaced objects factory.
            // Pass the instance of the class constructor argument as the parameter of the fa
            var substObject = Terrasoft.Core.Factories.ClassFactory.Get<UsrOriginalClass>(new

            // Retrieve the GetAmount() method output. Pass the values of the page input fiel
            int result = substObject.GetAmount(value1, value2);

            // Display the results on the page.
            return string.Format("The result value, retrieved after calling the replaceable c
            */

            /* Create a replacing class instance using the new() operator. */
            var substObjectByNew = new UsrOriginalClass();

            /* Create a replacing class instance using the replaced objects factory. */
            var substObjectByFactory = Terrasoft.Core.Factories.ClassFactory.Get<UsrOriginalC

            /* Retrieve the GetAmount() method output. Do not use replacement, call the UsrOr
            int resultByNew = substObjectByNew.GetAmount(value1, value2);

            /* Retrieve the GetAmount() method output. Call the method of the SubstituteClass
            int resultByFactory = substObjectByFactory.GetAmount(value1, value2);

            /* Display the results on the page. */
            return string.Format("Result without class replacement: {0}; Result with class re
        }
    }
}
```

The code provides examples of creating a replacing class both using the replaced objects factory and using

the `new()` operator.

6. Click [*Save*] then [*Publish*] on the Designer's toolbar.

As a result, Creatio will add the custom `UsrAmountService` REST web service that has the `GetAmount` endpoint.

Outcome of the example

To call the custom web service, access the `GetAmount` endpoint of the `UsrAmountService` web service in the browser and pass 2 arbitrary numbers as `value1` and `value2` parameters.

Request string

```
http://mycreatio.com/0/rest/UsrAmountService/GetAmount?value1=25&value2=125
```

The `GetAmountResult` property will return the custom web service results executed both using and not using the class replacement.