http://doi.org/10.35784/iapgos.8488

received: 30.09.2025 | revised: 10.12.2025 | accepted: 18.12.2025 | available online: 20.12.2025

IMPLEMENTING TRAITS IN C# USING ROSLYN SOURCE GENERATORS

Mykhailo Pozur[1], Viktoriia Voitko[1], Svitlana Bevz[2], Serhii Burbelo[2], Olena Kosaruk[1]

[1]Vinnytsia Technical National University, Vinnytsia, Ukraine, [2]Zhytomyr Military Institute named after S. P. Korolev, Zhytomyr, Ukraine

Abstract. *The paper presents an approach to implementing traits functionality in the C# programming language through the use of Roslyn Source Generators, a compile-time metaprogramming tool introduced in .NET5. Traits, which are composable units of fine-grained reuse, can benefit the C# language by bringing a mechanism for behavior reuse in the context of single inheritance. The article describes the design and implementation of a source generator that allows to annotate classes with metadata attributes, automatically injecting the corresponding member definitions at compile time. To achieve better code generation performance at design time, Incremental Source Generators were used. The composition of trait members is described in detail, focusing on name conflict resolution and type safety by utilizing interfaces and the override mechanism. Defining non-public members in a context of a trait is also discussed and implemented, thus allowing for better code reusability. The article concludes that language-level features such as traits can be implemented in the C# language by using Roslyn Source Generators, thus demonstrating the potential of such an instrument and metaprogramming in general.*

Keywords: C#, traits, object-oriented programming, metaprogramming, code generation

IMPLEMENTACJA CECH W JĘZYKU C# PRZY UŻYCIU ROSLYN SOURCE GENERATORS

Streszczenie. *Artykuł przedstawia podejście do implementacji funkcjonalności cech w języku programowania C# poprzez wykorzystanie Roslyn Source Generators, narzędzia do metaprogramowania w czasie kompilacji wprowadzonego w .NET5. Traity, które są komponowalnymi jednostkami drobnoziarnistego ponownego użycia, mogą przynieść korzyści językowi C# poprzez wprowadzenie mechanizmu ponownego użycia zachowania w kontekście pojedynczego dziedziczenia. W artykule opisano projekt i implementację generatora źródłowego, który umożliwia dodawanie adnotacji do klas za pomocą atrybutów metadanych, automatycznie wstrzykując odpowiednie definicje elementów członkowskich w czasie kompilacji. Aby osiągnąć lepszą wydajność generowania kodu w czasie projektowania, zastosowano przyrostowe generatory źródeł. Szczegółowo opisano skład członków cech, koncentrując się na rozwiązywaniu konfliktów nazw i bezpieczeństwie typów poprzez wykorzystanie interfejsów i mechanizmu zastępowania. Omówiono i zaimplementowano również definiowanie niepublicznych członków w kontekście cechy, co pozwala na lepsze ponowne wykorzystanie kodu. Artykuł podsumowuje, że funkcje na poziomie języka, takie jak cechy, można zaimplementować w języku C# za pomocą Roslyn Source Generators, demonstrując w ten sposób potencjał takiego narzędzia i ogólnie metaprogramowania.*

Słowa kluczowe: C#, traits, programowanie obiektowe, metaprogramowanie, generowanie kodu

Introduction

Inheritance is a fundamental mechanism in object-oriented programming, enabling the hierarchical organization of classes and a reuse of behavior. While both single and multiple inheritance paradigms have been proposed and implemented in various programming languages, single inheritance has achieved broader adoption in mainstream languages such as Java or C#. The preference for a single inheritance is attributed to the fact that it favors simplicity and predictability. On the other side, multiple inheritance allows for a subclass to inherit features from more than one superclass, which helps achieve better code reusability [1]. However, this comes at the cost of significant semantic complexity. One of the most well-known complications of multiple inheritance is the diamond problem, where a class inherits from two classes that share a common ancestor [8]. Resolving method conflicts and determining a coherent method resolution order in such cases can become non-trivial and language-dependent.

The limitations of a single inheritance led to the creation of alternative mechanisms for behavior reuse. Most notable are traits [19] (in PHP [11], Rust [23]), and mixins [20] (in Ruby [15], Dart [25]) providing compositional tools that decouple code reuse from the inheritance hierarchy. These constructs support multiple inheritance of behavior without a need for a direct multiple inheritance. For instance, traits promote horizontal reuse of methods while avoiding ambiguous method lineage, thereby preserving the benefits of multiple inheritance with fewer drawbacks [2].

In the context of C# programming language traits can be partially recreated by using default interface methods [14]. Interfaces allow defining not only behavior but also states (via property declarations). The main limitation of interfaces in context of composition is their lack of ability to define non-public members. Also, interfaces do not support some features of traits like aliasing and member exclusion [19]. C# programming language also supports mixin-style behavior composition via extension methods. Such approach is mostly used to extend functionality of base or imported types. Unlike interfaces extension method can't be composed into several types.

Thus, there is a need to improve capabilities of C# programming language in area of behavior composition. This can be achieved by implementing traits using metaprogramming techniques, such as code generation. Potential candidate to implement such functionality would be Roslyn Source Generators [13], which in combination with Roslyn API proven to be a powerful metaprogramming tool that can be used to implement language-level features [3, 12, 21].

1. Related works

The concept of traits was originally introduced in the context of the Smalltalk [5] programming language and has been extensively studied as a flexible mechanism for code reuse and composition, providing fine-grained control over behaviour injection without the drawbacks of classical multiple inheritance. Formal semantics and practical utility of traits as a mechanism separate from inheritance were established in papers [7, 19]. These ideas have since influenced experimental and production languages including Rust and PHP, each integrating traits with varying degrees of static safety and conflict resolution mechanisms.

Although the original traits definition describes them as a "set of methods, divorced from any class hierarchy" [19], several studies were made to introduce states to traits [4, 22]. The concept of traits with state has been more extensively explored in some programming languages. For example, Scala allows traits to contain both fields and methods, with clear linearization rules to resolve conflicts [24]. Similar ideas are present in Rust's traits combined with default method implementations and associated types, although Rust explicitly separates data from traits, requiring composition via structs [17].

Traits' impact on code reusability also was studied. For example, paper [6] describes implementing a stream library using traits. It evaluates the reusability of traits, describes problems that can be encountered while using them and reviews alternative composition mechanisms. Authors conclude that traits are a good concept for increasing code reusability, as using traits allowed them to reduce the amount of code by almost 40% in comparison to implementation without traits.

With the introduction of Roslyn Source Generators [13], a new compile-time metaprogramming model was introduced, offering a safer and more transparent alternative. Source Generators allow generating code at compile time and add it to the compilation. They are able to read the contents of the compilation before running, as well as access any additional files. This ability enables them to introspect both user C# code and generator-specific files. Source Generators don't enable any scenario that wasn't possible before. Their key benefit lies in the fact that they are built into Roslyn compiler. This allows for better integration with IDE, i.e. Source Generators are treated by IDE as code analyzers which allows for their execution at design time.

Along with the possibility to view generated artifacts at design time, this feature brings one significant drawback. As an input Source Generators receive an object model representing the code being compiled, which means they need to run every time the source code is changed. To address this issue, Microsoft came up with Incremental Source Generators [10]. Such type of generators uses a pipeline to filter and transform input before generation. A key feature of such pipeline is its ability to cache the output of each step and use memoization to significantly reduce the number of calculations and generations. Basically, if there is an output value that is present in the cache, then the generator uses already calculated and generated result instead of proceeding with calculations and generation. Such an approach enables using incremental source generators for projects with a large code base.

Being an integral part of a .NET Compiler Platform SDK allows Source Generators to benefit from using Roslyn API, which allows for advanced code analysis and transformations. For example, article [16] describes developing a Roslyn-based optimization engine to enhance the performance and security of cloud-hosted .NET applications. This research demonstrates the power of Roslyn's advanced capabilities for code analysis and refactoring.

In terms of using code generation several community-driven efforts, such as the StrongInject [21], Immutype [12], and AutoInterface [3] projects have demonstrated how source generators can automate code patterns such as dependency injection, immutability enforcement, and interface implementation. This demonstrates that Source Generators can be used to implement language-level features.

2. Proposed methodology

For traits implementation in the C# programming language, we defined a model based on the one proposed in the paper [19]. This model operates with next sets:

- N_m, set of method identifiers,
- B_m, set of method definitions,
- N_s set of state identifiers,
- B_s, set of state definitions.

This way method can be defined as $a \longmapsto m$, where $a \in N_m, m \in B_m$. Same goes for state, which can be defined as $b \longmapsto s$, where $b \in N_s, s \in B_s$. As methods and states can be undefined or have conflicts, extended sets are defined: B_m^* and B_s^*. Those sets contain two additional definitions: $\mathbb{U}$ which defines undefined method or state and $\mathbb{C}$ wich defines confilct. States and methods are separated due to the differences in their indentifiers and definitions. For state identifier is simply a name but for method it also includes parameters. As for definition, method definition is it's body and for state its a bit more complicated. It is impossible to check if states have identical definitions because their definition is in fact their purpose which can only be known by a developer. The only reasoning behind having defitions for states – to have the ability to use aliases for them.

Let's define methods as dictionary $d \in D$, $d: N_m \rightarrow B_m^*$, where $d^{-1}(B_m)$ is finite set and $d^{-1}(\mathbb{C}) = \emptyset$. And define states in the same way $v \in V$, $v: N_s \rightarrow B_s^*$ where $v^{-1}(B_s)$ is finite set and $v^{-1}(\mathbb{C}) = \emptyset$. This way class $c \in C$ can be defined as $c = \langle v, d \rangle$. Trait $t \in T$ can be defined in a same way $t = \langle v, d \rangle$.

Sum of dictionaries can be defined as

$$(d_1 + d_2)(l) = d_1(l) \sqcup d_2(l) \qquad (1)$$

Operation $\sqcup$ for each set of definitions (B_m and B_s) giving that $m_1 \neq m_2$, can be defined as:

$\sqcup$	$\mathbb{U}$	m_1	m_2	$\mathbb{C}$
$\mathbb{U}$	$\mathbb{U}$	m_1	m_2	$\mathbb{C}$
m_1	m_1	m_1	$\mathbb{C}$	$\mathbb{C}$
m_2	m_2	$\mathbb{C}$	m_2	$\mathbb{C}$
$\mathbb{C}$	$\mathbb{C}$	$\mathbb{C}$	$\mathbb{C}$	$\mathbb{C}$

Override is an important part of name conflict resolution between methods and is defined as:

$$(d \rhd d_t)(l) = \begin{cases} d_t(l) & d(l) = \mathbb{U} \\ d(l) \end{cases} \qquad (2)$$

Such operation can only be applied to method dictionaries. Next operation that can help in name conflict resolution is aliasing and is defined as:

$$d_t[a \rightarrow b](l) = \begin{cases} d_t(l) & l \neq a \\ d_t(b) & l = a \land d_t(a) = \mathbb{U} \\ \mathbb{C} \end{cases} \qquad (3)$$

This operation can be applied to both methods and states. But such operation results in creation of a new dictionary record. To avoid conflicts the original identifier needs to be removed. This requires usage of an exclusion operation:

$$(d_t - a)(l) = \begin{cases} \mathbb{U} & a = l \\ d_t(l) \end{cases} \qquad (4)$$

In paper [19], it is suggested to use override operation for all methods on trait composition. But it can lead to number of implicit overrides. To overcome this, we define set of trait methods that can be overridden on composition $o_t \in O$, $o_t: N_m \rightarrow B_m^*$ where $o_t^{-1}(\mathbb{U}) = \emptyset$ and $o_t^{-1}(B_m)$ is finite. This way trait can be defined as $t = \langle v_t, d_t, o_t \rangle$ where $(d_t + o_t)^{-1}(\mathbb{C}) = \emptyset$. Thus, operation of trait composition into class is defined:

$$c \cup t = \langle v + v_t, (d \rhd o_t) + d_t \rangle \qquad (5)$$

This way we can control which methods are allowed to be overridden on composition.

Our method of implementing traits in the C# programming language relies upon using code generation tools to achieve desired results. Compile-time metaprogramming is better suited for this task for multiple reasons:

- runtime metaprogramming in .NET does not allow type modifications; it only allows creating new types or lambda functions,
- compile-time metaprogramming does not introduce any runtime overhead,
- compile-time metaprogramming allows our solution to benefit from static code analysis, which can improve type safety and help with name conflicts resolution,
- compile-time and/or design-time generation allows to see composition results before compilation.

There are two main candidates to implement such generation: T4 Templates and Roslyn Source Generators. We've chosen Source Generators due to number of reasons:

- Source Generators are part of Roslyn meaning they have access to compile-time metadata on a same level as compiler does,
- Source Generators are embedded on analyzer level meaning they can generate code at design-time,

- design-time generation will allow developers to benefit from various IDE features like IntelliSense or other syntax highlight features,
- Incremental Source Generators [10] allow to build efficient code generation that relies on smart node selection, caching and memoization to reduce number of computations and generations.

T4 Templates despite being a powerful metaprogramming tool lacks certain capabilities of Source Generators, especially in area of design-time code generation.

The main issue with the suggested approach is that such implementation will not be done on a level of programming language but rather as a 3rd party library. This leads to several issues that need to be overcome.

The first issue is finding an alternative to programming language keywords, as languages that implement traits use keywords to define traits and control the behavior of their members. The only alternative we deem feasible is using metadata attributes. Such attributes are easy to assign and access during code generation using Roslyn API and can be accessed in a runtime with the help of Reflection.

As mentioned before, other languages utilize dedicated keywords to mark a certain code element as a trait declaration, thus making a trait a separate type of syntax node. In the case of C# language, two options can act as a "trait". The most obvious one is the interface as it allows declaring member implementations by using default method implementations [14]. However, using them will limit traits to public members only and will not allow declaring fields. The next option would be using abstract classes to define traits. This enables defining non-public members with the added benefit of declaring fields. Also, abstract classes are designed to have implementations and states, while interfaces are designed to provide a list of methods implemented by a class. It is also worth mentioning that by design traits need to be "divorced from any class hierarchy" [19], meaning they can't inherit other types or be inherited themselves. This way we can define trait as:

- an abstract class that can't inherit any other classes or be inherited,
- is assigned a predefined metadata attribute that marks such class as a trait,
- does not declare constructor.

The next issue is related to the way how a trait can be composed into a class. This issue arises because Source Generators by design are additive only, meaning they can only add a new code but can't modify an existing one. This way class can be extended only by adding a new source code file. Thankfully, C# programming language allows to overcome this issue by using partial class declarations [18]. It allows a class to be declared throughout multiple source files, but to be able to do so, the class needs to be declared as "partial".

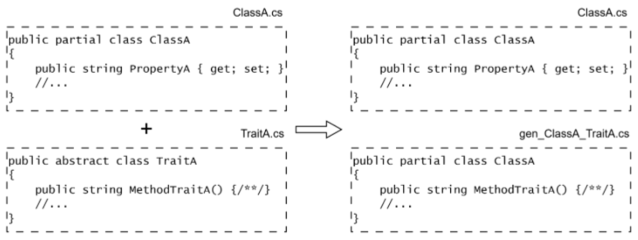
The trait is composed into a class by generating a partial class declaration containing members of a corresponding trait. We call a class that is composing a trait a "target class" as it is a target to which trait's members are copied.

Key benefits of such implementation are:

- trait members are composed directly into a class, thus removing a need to handle nested classes or complex type hierarchy,
- trait in this case is a part of a syntax tree, meaning it benefits from static code analysis and other IDE features,
- as Source Generators are executed at design time, all composed members are visible by code analyzers and IDE without a need to run compilation.

But such implementation has several shortcomings:

- the generator needs to access both trait and target class declarations, meaning that trait can't be defined in a 3rd party library (i.e. NuGet),
- as a trait member is copied into a class declaration, reference to that member's original declaration is lost.



```
public partial class ClassA
{
    public string PropertyA { get; set; }
    //...
}
```

```
public abstract class TraitA
{
    public string MethodTraitA() {/**/}
    //...
}
```

```
public partial class ClassA
{
    public string PropertyA { get; set; }
    //...
}
```

```
public partial class ClassA
{
    public string MethodTraitA() {/**/}
    //...
}
```

Fig. 1. Example of trait and class composition

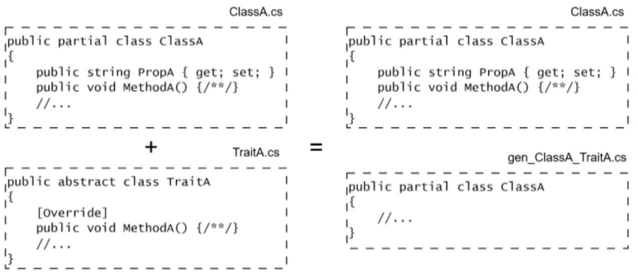
3. Review of traits implementation with code generation

The major part of implementing traits in C# with code generation is defining how individual members are composed into a class. To define that, first we must know what features traits will support in this implementation. Our idea for an initial set of features was:

- traits need to support member override,
- traits can implement interfaces,
- traits can define a set of members that represent a "contract" that the target class needs to implement,
- traits can have states.

Member override is an essential instrument to name conflicts resolution between trait and target class. A key use case scenario is when a trait and a class that implements it have members with the same names. This way, when composing trait members into a class, conflict of names will arise. To solve this problem, the traits model suggests using an override mechanism. We implement such a mechanism in a way that a trait member that has a name conflict with a member in a target class is not composed into that class.

We also want to introduce more control over such process by allowing specifying which members are allowed to be overridden by introducing Override metadata attribute. This will help to avoid unexpected behaviour by removing implicit overrides.



```
public partial class ClassA
{
    public string PropA { get; set; }
    public void MethodA() {/**/}
    //...
}
```

```
public abstract class TraitA
{
    [Override]
    public void MethodA() {/**/}
    //...
}
```

```
public partial class ClassA
{
    public string PropA { get; set; }
    public void MethodA() {/**/}
    //...
}
```

```
public partial class ClassA
{
    //...
}
```

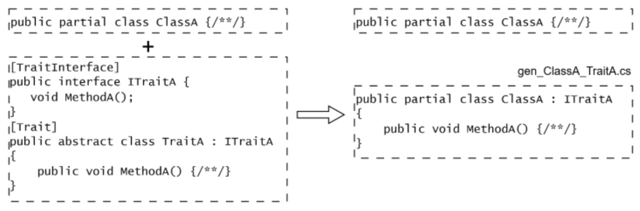
Fig. 2. Composition of a trait with a member override

In the given example (Fig. 2) both the trait and the target class contain methods with identical names (MethodA) and this method is marked with an Override metadata attribute in a trait. This way trait member is not composed into a target class.

Allowing traits to implement interfaces can help in a number of ways. First, it can enable using pattern matching to detect if a class has a specific trait as a part of its composition. Second, it allows defining a contract that the trait needs to implement. Also, interfaces can be utilized to define a contract that the trait expects to be implemented by the target class. Both types of contracts require different member behavior on composition.

Let's start with the trait contract first. To define such a contract, we introduce the metadata attribute "TraitContract" which is used to mark interfaces that are used as contract definitions for a trait. This way we can clearly define the purpose of an interface. Implemented members of such interfaces will be composed into the target class. Also, the trait contract interface needs to be "inherited" by the target class to indicate that class is composing the corresponding trait (see Fig. 3).

Although traits that implement interfaces have a number of advantages, we believe that such a way of defining traits should not be mandatory. So, we still allow a trait without an interface to exist and be composed into a class.



```
public partial class ClassA {/**/}
+
[TraitInterface]
public interface ITraitA {
  void MethodA();
}
[Trait]
public abstract class TraitA : ITraitA
{
    public void MethodA() {/**/}
}
```

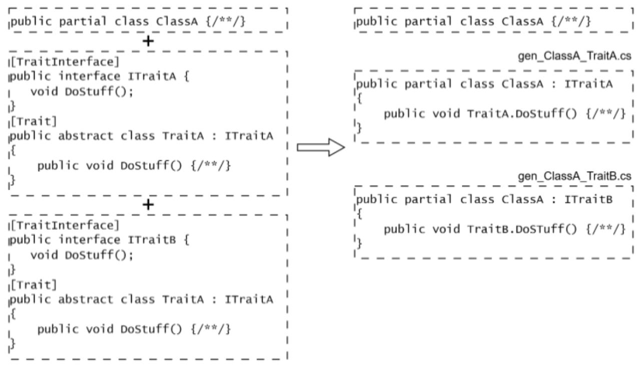
```
public partial class ClassA {/**/}
```

gen_ClassA_TraitA.cs

```
public partial class ClassA : ITraitA
{
    public void MethodA() {/**/}
}
```

Fig. 3. Composition of a trait that implements contract defined by interface

Introducing trait interfaces can also help with a certain case of name conflict. If override can help to resolve name conflicts that occur between trait and target class members, interfaces can help with conflicts between different traits. This can be achieved by using explicit interface implementations [9]. It allows to add an interface specifier in front of a member that is part of an interface.



```
public partial class ClassA {/**/}
+
[TraitInterface]
public interface ITraitA {
  void DoStuff();
}
[Trait]
public abstract class TraitA : ITraitA
{
    public void DoStuff() {/**/}
}
+
[TraitInterface]
public interface ITraitB {
  void DoStuff();
}
[Trait]
public abstract class TraitA : ITraitA
{
    public void DoStuff() {/**/}
}
```

```
public partial class ClassA {/**/}
```

gen_ClassA_TraitA.cs

```
public partial class ClassA : ITraitA
{
    public void TraitA.DoStuff() {/**/}
}
```

gen_ClassA_TraitB.cs

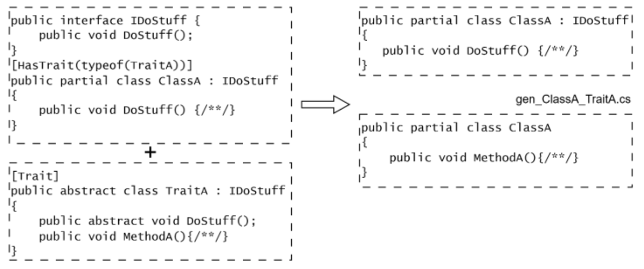
```
public partial class ClassA : ITraitB
{
    public void TraitB.DoStuff() {/**/}
}
```

Fig. 4. Usage of an explicit interface implementation to resolve name conflicts

Figure 4 gives an example of such a scenario. In this case, we have two traits each of them implementing a corresponding interface. Both interfaces and traits define members with the same name (method DoStuff). As member is defined within both interfaces it enables using explicit interface implementation. By adding an interface specifier in front of each implementation we are able to keep both method definitions in a target class.

The main drawback of this approach is that such member definitions can be accessed only when the class is cast to a type of interface. Because of this we also introduced a metadata attribute to mark members which are allowed to be defined in a target class in a such way.

The next way of using interfaces with traits is defining a contract that the trait expects to be implemented by the target class. This will allow to define a set of members via interface that can be then used within a trait and are present in a target class.



```
public interface IDoStuff {
    public void DoStuff();
}
[HasTrait(typeof(TraitA))]
public partial class ClassA : IDoStuff
{
    public void DoStuff() {/**/}
}
+
[Trait]
public abstract class TraitA : IDoStuff
{
    public abstract void DoStuff();
    public void MethodA(){/**/}
}
```

```
public partial class ClassA : IDoStuff
{
    public void DoStuff() {/**/}
}
```

gen_ClassA_TraitA.cs

```
public partial class ClassA
{
    public void MethodA(){/**/}
}
```

Fig. 5. Usage of interfaces to define a contract for a target class in a trait

As can be seen in the example (Fig. 5) such interfaces are not assigned any metadata attribute to mark them as a contract designed to be used by a trait. This way our implementation automatically treats any interfaces that are not marked as "trait contract" to be a contract for a target class. This is done primarily because classes can implement interfaces that belong to the different context. We want to enable our traits to target such implementations without a need to define a new interface or interfere with an existing one. Also, this enables using traits for classes that implement interfaces from 3rd party libraries.

This way of implementing contracts for target classes has several advantages:

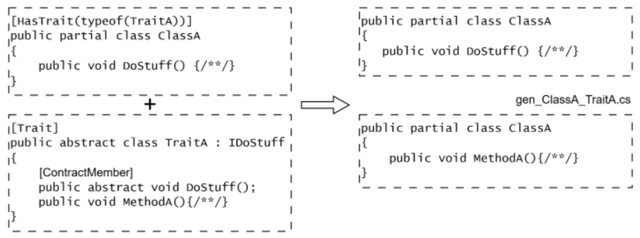
- it relies upon an existing language feature,
- it enables to definition of a strict contract which is guaranteed to be implemented by the static analyzer and compiler,
- it allows defining traits that can target a wide variety of classes.

And most notable drawbacks of such an approach are:

- higher complexity of type hierarchy,
- need to define all interface members within a trait.

The last drawback can be overcome by generating a partial declaration of the trait containing interface members' declarations. Thus, removing a need to manually define such members.

Another way of introducing contracts to traits would be using "placeholders" to mock members of a target class. It can be done by utilizing metadata attributes to mark such members. We call such contracts "weak" because they are not strictly defined by a language mechanism.



```
[HasTrait(typeof(TraitA))]
public partial class ClassA
{
    public void DoStuff() {/**/}
}
+
[Trait]
public abstract class TraitA : IDoStuff
{
    [ContractMember]
    public abstract void DoStuff();
    public void MethodA(){/**/}
}
```

```
public partial class ClassA
{
    public void DoStuff() {/**/}
}
```

gen_ClassA_TraitA.cs

```
public partial class ClassA
{
    public void MethodA(){/**/}
}
```

Fig. 6. Composition of a trait with weak contract

Example (Fig. 6) demonstrates defining the "placeholder" member (method DoStuff) by using metadata attribute (ContractMember). Such a member can be referenced within a trait in the same way as a member of an interface contract. In terms of trait member composition, such members are not composed into a target class.

The main advantages of such approach are:

- no need to rely on an interface to access target class members,
- results in a simpler hierarchy,
- more flexibility in contract definition,
- allows accessing private or protected members of a target class.

In terms of shortcoming, the most notable are:

- contract implementation is guaranteed only by syntax errors in case of mismatch,
- such a contract can only be defined in the context of a single trait and can't be shared between other traits,
- the target class is not aware of such contact.

Despite "weak" contacts providing an easier way of accessing target class's members, we strongly recommend using interfaces for such purpose as they provide way more reliable way of doing so. We recommend using "weak" contacts only in cases when defining interfaces is very inconvenient or trait needs to access private or protected members.

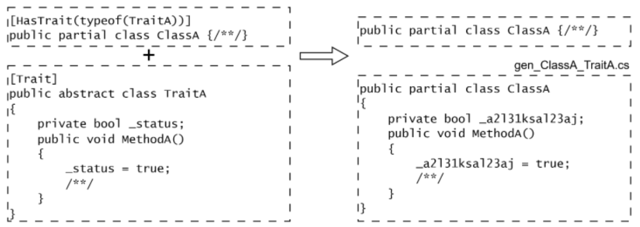
Next, we decided to go further and experimented with private and protected members in traits. The key idea behind it was to allow defining private or protected members in traits as such feature can be used in a number of scenarios. Most notable are:

- defining "placeholders" to access private or protected members of target classes via "weak" contract,
- defining "local" members which are only relevant in the context of a trait,
- expanding the target class with non-public members.

The idea of traits having so-called "local" members comes from the principle of encapsulation in OOP, as classes can define private members that are accessible only in the context of that class's declaration. So, we decided to replicate similar behavior for traits.

The key problem with trait's local members arises from the nature of our trait's implementation. As trait's members are being copied into the target class, its members become members of the target class. This way trait's private members become private members of the class after composition, meaning they can be accessed in the context of that class.

To overcome this issue, we suggest using obfuscation techniques to change names of trait's private members on composition. The new name can be assigned on each code generation or compilation by using pseudo-random number generation. Such a member is still technically accessible within a target class, but doing so becomes almost impossible.



```
[HasTrait(typeof(TraitA))]
public partial class ClassA {/**/}
```

+

```
[Trait]
public abstract class TraitA
{
    private bool _status;
    public void MethodA()
    {
        _status = true;
        /**/
    }
}
```

```
public partial class ClassA {/**/}
```

gen_ClassA_TraitA.cs

```
public partial class ClassA
{
    private bool _a2131ksal23aj;
    public void MethodA()
    {
        _a2131ksal23aj = true;
        /**/
    }
}
```

Fig. 7. Composition of a trait with a private member

As can be seen in the example (Fig. 7), the private member was composed into a target class with a new name. Note, that the member is renamed in the context of the whole trait, meaning that all references are also renamed.

Such an approach does not only allow to "encapsulate" trait members, but also helps with a name conflict resolution due to the random nature of the member's name. This way several traits can define private members with an identical name and still be composed into a class without name conflict occurring.

In terms of code generation proposed implementation relies upon using Incremental Source Generators. Utilizing such generators allows for achieving efficient design time code generation by taking advantage of their caching and memoization mechanism. A key aspect of achieving efficient generation is pipeline design. The incremental generator pipeline needs to select only relevant data and present it in a way that can be used by the caching mechanism.

Let's review such a pipeline (Fig. 8) in detail. It uses SyntaxProvider as its primary data source as the generator needs to access class and trait declarations in a source code. Next, the pipeline filters out syntax nodes that are class declarations and are assigned the "HasTrait" metadata attribute. This is done by using the ForAttributeWithMetadataName pipeline initialization call. After the initial filtering of syntax nodes, they are transformed into collections of class-trait pairs.

The process of forming class-trait pairs is complex and consists of multiple steps:

1) Information about trait types that are composed into the class using the metadata attribute is retrieved.
2) For each trait corresponding object containing all information required for code generation is created.
3) The object describing information about the target class is created.
4) Name conflicts are checked and resolved if possible.
5) Collection of trait-class pairs is formed.

The object describing trait's data includes:

- information about all members and metadata attributes assigned to them,
- information about interfaces implemented by a trait,
- information about namespaces used in trait declaration,
- Roslyn type metadata (ITypeSymol) of a trait,
- reference to the syntax of a trait's declaration.

Such transformation results in a collection of class-trait pairs which is not ideal for Incremental Source Generators as they generate code in the context of the output unit. For example, if the pipeline outputs all data as a single collection (i.e. 2d array with dimensions [1, n]), then generation will happen in the context of all data. The generator still be able to produce a file per item in the array, but such generation happens for the whole output, meaning caching is not used properly. To overcome this, data needs to be flattered into a 1d array. This can be done by using the SelectMany operation of the pipeline. This way generation will happen in the context of an individual class-trait pair, which enables better caching utilization.

The last step of the pipeline before generation is assigning a custom comparer. This is required as we use custom data types to store information about class-trait pairs. So, we define custom comparer which allows us to set rules for data comparison by caching mechanism.

For source code generation we rely primarily on Roslyn API instead of template-based code generation. This way we are able to work with source code as data, which enables usage of IDE-like features (such as renaming a member, finding all references, etc.). Also, this way we are able to significantly reduce the chances of generating a code with syntax errors.

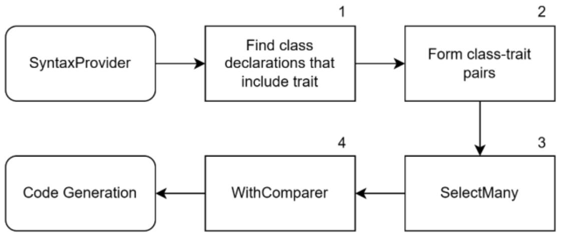
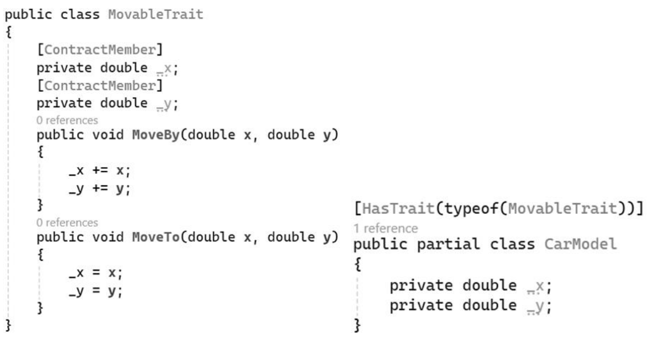


Fig. 8. Incremental generator pipeline for traits code generation

4. Experiments

Purpose of an experiment is to validate that our implementation behaves as described. First, we want to check access to private members of the target class. Let's define a class and a corresponding trait (Fig. 9).



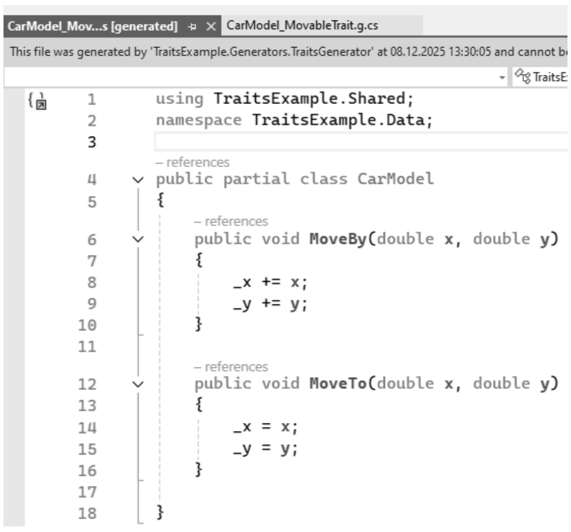
```
public class MovableTrait
{
    [ContractMember]
    private double _x;
    [ContractMember]
    private double _y;
    0 references
    public void MoveBy(double x, double y)
    {
        _x += x;
        _y += y;
    }
    0 references
    public void MoveTo(double x, double y)
    {
        _x = x;
        _y = y;
    }
}
```

```
[HasTrait(typeof(MovableTrait))]
1 reference
public partial class CarModel
{
    private double _x;
    private double _y;
}
```

Fig. 9. Source code of a class and a trait that access private members of its target class

As can be seen in example (Fig. 9), trait requires its target to defined private states ("_x" and "_y"). Generated code for such scenario should contain only methods "MoveBy" and "MoveTo" (Fig. 10).

IntelliSense is also picking up generated methods for "CarModel" class (Fig. 11).



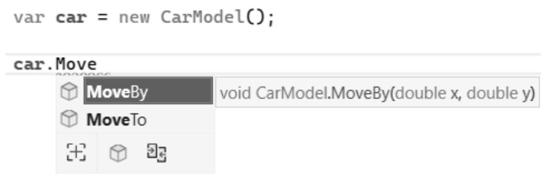
This file was generated by 'TraitsExample.Generators.TraitsGenerator' at 08.12.2025 13:30:05 and cannot b

```
using TraitsExample.Shared;
namespace TraitsExample.Data;

– references
public partial class CarModel
{
    – references
    public void MoveBy(double x, double y)
    {
        _x += x;
        _y += y;
    }

    – references
    public void MoveTo(double x, double y)
    {
        _x = x;
        _y = y;
    }
}
```

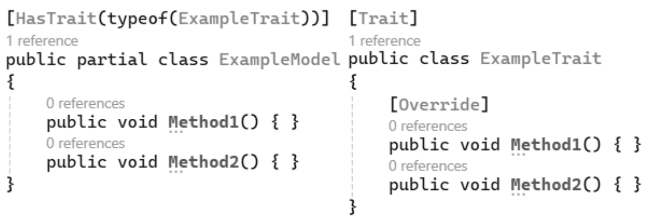
Fig. 10. Generate code for test scenario when the trait has to access private members of its target class



```
var car = new CarModel();

car.Move
```

Fig. 11. IntelliSense suggestions for CarModel class which includes MovableTrait

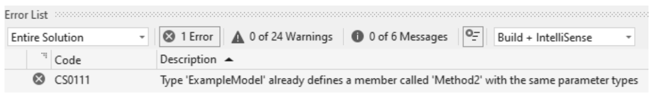


```
[HasTrait(typeof(ExampleTrait))]
1 reference
public partial class ExampleModel
{
    0 references
    public void Method1() { }
    0 references
    public void Method2() { }
}
```

```
[Trait]
1 reference
public class ExampleTrait
{
    [Override]
    0 references
    public void Method1() { }
    0 references
    public void Method2() { }
}
```

Fig. 12. Source code of a method override test

An important part of our implementation is method override handling. Let's define a class and a trait that both contain two methods with identical names, one of which can be overridden on compositions (Fig. 12). Result of such composition should be syntax error due to name conflict (Fig. 13) as "Method2" is not defined as such that can be overridden on composition.



Code	Description
CS0111	Type 'ExampleModel' already defines a member called 'Method2' with the same parameter types

Fig. 13. Syntax error example due to name conflict

5. Conclusions

The prototype library that enables using traits in C# language was developed. The library was tested in a set of scenarios including using it in a medium-scale commercial project. All tests showed that the library works as intended.

The current set of features, implemented in the library includes:

- declaring traits and composing their members into classes,
- defining methods that can be overridden in a target class,
- using explicit interface member implementation to resolve name conflicts between traits,
- defining strict contacts for a trait and target class using interfaces,
- defining "weak" contract for a target class using metadata attributes,
- declaring non-public members.

The proposed method of implementation allows achieving a wider range of functionality in comparison to existing C# mechanisms for behavior composition, which enables using it in a broader range of scenarios. This helps to improve code reusability and modularity by introducing a better mechanism of fine-grained reuse. On top of that, this paper demonstrates the power of Incremental Source Generators and metaprogramming in general. Utilizing code generation in conjunction with Roslyn API allowed the implementation of a language-level feature.

References

[1] Albalooshi F., Mahmood A.: A Comparative Study on the Effect of Multiple Inheritance Mechanism in Java, C++, and Python on Complexity and Reusability of Code. Int. J. Adv. Comput. Sci. Appl. 8(6), 2017, 109–116 [https://doi.org/10.14569/ijacsa.2017.080614].
[2] Arora H., Servetto M., Oliveira B. C. D. S.: Separating Use and Reuse to Improve Both. The Art, Science, and Engineering of Programming 3(3), 2019 [https://doi.org/10.22152/programming-journal.org/2019/3/12].
[3] AutoInterface, 2025 [https://github.com/beakona/AutoInterface].
[4] Bergel A. et al.: Stateful Traits. De Meuter W. (eds): Advances in Smalltalk. ISC 2006. Lecture Notes in Computer Science 4406. Springer, Berlin, Heidelberg 2007 [https://doi.org/10.1007/978-3-540-71836-9_4]
[5] Black A. P., Schärli N., Ducasse S.: Applying Traits to the Smalltalk Collection Classes. ACM SIGPLAN Not. 38(11), 2003, 47–64 [https://doi.org/10.1145/949343.949311].
[6] Cassou D., Ducasse S., Wuyts R.: Traits at work: The design of a new trait-based stream library. Comput. Lang., Syst. & Struct. 35(1), 2009, 2–20 [https://doi.org/10.1016/j.cl.2008.05.004].
[7] Ducasse S. et al.: Traits: A mechanism for fine-grained reuse. ACM Trans. Program. Lang. Syst. 28(2), 2006, 331–388 [https://doi.org/10.1145/1119479.1119483]
[8] Ducournau R., Morandat F., Privat J.: Empirical assessment of object-oriented implementations with multiple inheritance and static typing. ACM SIGPLAN Not. 44(10), 2009, 41–60 [https://doi.org/10.1145/1639949.1640093].
[9] Explicit Interface Implementation – C#, 2022 [https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/explicit-interface-implementation].
[10] Gerr P.: Incremental Roslyn Source Generators In .NET 6: Code Sharing Of The Future – Part 1. 2022 [https://www.thinktecture.com/net/roslyn-source-generators-introduction/].
[11] Hossain M.: Understanding Traits in PHP: A Guide with Examples, 2024 [https://mohasin-dev.medium.com/understanding-traits-in-php-a-guide-with-examples-84f14f1b7b43].
[12] Immutype, 2025 [https://github.com/DevTeam/Immutype].
[13] Introducing C# Source Generators, 2025 [https://devblogs.microsoft.com/dotnet/introducing-c-source-generators/].
[14] Lock A.: Understanding C# 8 default interface methods. 2024 [https://andrewlock.net/understanding-default-interface-methods/].
[15] Mr Prime: Mixins in Ruby, 2024 [https://medium.com/@primedruk312/mixins-in-ruby-09e58b226e50].
[16] Muniyandi V.: Harnessing Roslyn for Advanced Code Analysis and Optimization in Cloud- Based .NET Applications on Microsoft Azure. Int. J. Commun. Netw. Secur. 14(3), 2022, 979–990 [https://ijcnis.org/index.php/ijcnis/article/view/8051/2231].
[17] Nichols C., Klabnik S.: Rust Programming Language, 2nd Edition. No Starch Press, Incorporated, 2023.
[18] Partial Classes and Members – C#, 2025 [https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/partial-classes-and-methods].
[19] Schärli N. et al.: Traits: Composable Units of Behaviour. In: ECOOP 2003 – Object-Oriented Programming. Springer Berlin Heidelberg, 2003, 248–274 [https://doi.org/10.1007/978-3-540-45070-2_12].
[20] Steyaert P. et al.: Nested Mixin-Methods in Agora. Nierstrasz O. M. (eds): ECOOP' 93 — Object-Oriented Programming. Springer Berlin Heidelberg, 1993 [https://doi.org/10.1007/3-540-47910-4_12].
[21] StrongInject, 2025 [https://github.com/YairHalberstadt/stronginject].
[22] Tesone P. et al.: A new modular implementation for stateful traits. Sci. Comput. Program. 195, 2020, 102470 [https://doi.org/10.1016/j.scico.2020.102470].
[23] Traits: Defining Shared Behavior, 2025 [https://doc.rust-lang.org/book/ch10-02-traits.html].
[24] Wampler D., Payne A.: Programming Scala. O'Reilly Media, Incorporated, 2014.
[25] Yaqoob E.: Mixins in dart, 2023 [https://medium.com/@emanyaqoob/mixins-in-dart-aed7e89de15d].

M.Sc. Mykhailo Pozur
e-mail: mixalchik545@gmail.com

Mykhailo Pozur is a Ph.D. student in software engineering at the Department of Software Engineering, Vinnytsia National Technical University, Vinnytsia, Ukraine.
Scientific interests of the leader: software engineering, metaprogramming, computer science, information technologies.

https://orcid.org/0009-0003-5225-2453

Ph.D. Viktoriia Voitko
e-mail: dekanfki@i.ua

Viktoriia Voitko is associate professor at the Department of Software Engineering, Vinnytsia National Technical University, Vinnytsia, Ukraine.
Scientific interests of the leader: software engineering, computer science, information technologies.

https://orcid.org/0000-0002-3329-7256

Ph.D. Svitlana Bevz
e-mail: bevz.svitlana.v@gmail.com

Svitlana Bevz is lecturer at the Department of Electrical Engineering and Electronics, Zhytomyr Military Institute named after S. P. Korolev, Zhytomyr, Ukraine.
Scientific interests of the leader: software engineering, computer science, information technologies.

https://orcid.org/0000-0002-4651-2453

Ph.D. Serhii Burbelo
e-mail: smburbelo@gmail.com

Serhii Burbelo is lecturer at the Department of Electrical Engineering and Electronics, Zhytomyr Military Institute named after S. P. Korolev, Zhytomyr, Ukraine.
Scientific interests of the leader: software engineering, computer science, information technologies.

https://orcid.org/0000-0002-8554-2292

Ph.D. Olena Kosaruk
e-mail: lena.kosaruk@vntu.edu.ua

Ph.D., assistant professor, Faculty of Management and Information Security, Vinnytsia National Technical University, Vinnytsia, Ukraine. Author and co-author of more than 30 scientific publications.
Scientific interests of the leader: fuzzy modeling, digital economy, dual education, development of soft and hard skills.

https://orcid.org/0000-0003-1346-2944