DOCUMENTATION

1. Table of content

1 Introduction

AspectDN aims to weave through advices:

- ✓ New types (class, enumeration, delegates, structures, interfaces),
- ✓ New members in a type or interface (method, field, event, constructors, properties, operators, indexers).
- ✓ New attributes to assemblies, types and members.
- ✓ New inheritance (interface or base type)
- ✓ Code snippets to methods, properties (accessors) and events (add and remove)

AspectDN is designed to encourage a weak coupling of advices with their target and thus allow the reuse of advices. Therefore, it offers the ability to describe the target context with prototype member or prototype type usable by the advices.

The purpose of this document is to describe **AspectDN** functionalities and the way to code pointcuts, advices, prototypes and aspects and to use the aspect weaver.

First, we will define what is an advice, a pointcut, a join point and an aspect.
Then, we will describe how to declare advice, pointcut and aspect.
At least, we will explain how to create an **AspectDN** project and how to perform a weaving.

All the syntax described in the document are for the moment issue from the C#5 AspectDN language.

2 Terms

2.1 Cross-cutting concerns

It is an identical function scattered in different places of the application

Example:
Security, logs....

2.2 Advice

This is an element that we want to apply to the existing model.

2.3 Pointcut

This is the term given to determine the place in the application where the plugin is inserted by the weaver
Example: a method, property, class....

2.4 Join point

This is the term given to determine where the plugin should be executed on the pointcut.
Example: before the method is called, before a field is update by a value.

2.5 Aspect

An aspect is the combination of a pointcut, join point and advice.

3 Package

3.1 Explanation

The package is used to declare a set of related items. You can use a package to organize advices, pointcuts and aspects. It has similar functionality to C# namespaces

Using another package is done by “using” as in C#. You can also reference an advice by its full name package_name.advice_name.

3.2 Syntax

```
aspect-compilation-unit:
       [using-directives] [package-declarations] [package-member-declarations]

package-declarations:
       package-declaration
       package-declarations package-declaration

package-declaration:
       'package' qualified-identifier package-body

package-body:
       '{' [using-directives]  [package-declarations] [package-member-declarations]   '}'

package-member-declarations:
       package-member-declaration
       package-member-declarations   package-member-declaration

package-member-declaration:
       pointcut-declaration
       advice-declaration
       aspect-declaration
       prototype-type-declaration
       prototype-mapping-types-declaration
```

3.3 Example

Package declaration with a pointcut

```
using System;
package foundation.p
{
       methods px : methods.Name == “methodName”;
}
```

Package declaration with an advice

```
using System;
package foundation.a
{
       advice code ax
       {
              System.Console.WriteLine("before");

              [around anchor];

              System.Console.WriteLine("after");
       }
}
```

Package declaration w

```
using System;
using foundation.p;
using foundation.a;
package foundation.aspect
{
```

```
        myAspect =>
                extend around call px with ax;
}
```

Or by defining everything in the aspect

```
using System;
package foundation.aspect
{
        myAspect =>
                extend around call foundation.p.px with foundation.a.ax;
}
```

4 Pointcut

4.1 Explanation

There are two types of pointcuts.

A named pointcut is a pointcut that can be used in any compatible aspect.
However, a pointcut can be defined locally in the aspect which will not be reusable in another aspect.

We can define different can of pointcuts:

- ✓ assemblies: targeting modules in assemblies
- ✓ classes: targeting class in assemblies
- ✓ interfaces: targeting interfaces in assemblies
- ✓ methods: targeting methods in assemblies
- ✓ fields: targeting fields in assemblies
- ✓ properties; targeting properties in assemblies
- ✓ events; targeting events in assemblies
- ✓ delegates: targeting delegates in assemblies
- ✓ structs: targeting structures in assemblies
- ✓ exceptions: targeting exceptions in assemblies
- ✓ constructors: targeting constructors in assemblies
- ✓ enums: targeting enumerations in assemblies

4.2 Syntax

```
pointcut-type identifier  : pointcut-expression ;

pointcut-expression:
        expression

pointcut-type:
        'assemblies'
        'classes'
        'interfaces'
        'methods'
        'fields'
        'properties'
        'events'
        'delegates'
        'structs'
        'exceptions'
        'constructors'
        'enums'
```

“pointcut-type” defines the kind of pointcut
“identifier” is the name of the pointcut that aspect has to reference.

« pointcut-expression » is a lambda expression. According to the kind of pointcut, the referenced member changes and their related members too. The related members have their corresponding members in the definition class of CECIL.

Pointcut kind	Referenced Member	Related Member (Cecil class)
assemblies	assemblies	ModuleDefinition
classes	classes	TypeDefinition
methods	methods	MethodDefinition
fields	fields	FieldDefinition
properties	properties	PropertyDefinition
events	events	EventDefinition
delegates	delegates	TypeDefinition
structs	structs	TypeDefinition
exceptions	exceptions	TypeDefinition
constructors	constructors	MethodDefinitions
enums	enums	TypeDefinition

Note
You will see later (see join point) that we can target a call of a method for instance. In this case, another Referenced Member should be used in addition of methods “caller” with related members corresponding to cecil class MethodDefinition.

4.3 Example

In the following example, the pointcut named “myPointcut” should target all methods of the targeting assemblies which has the name “MethodName”

```
methods myPointCut : methods.Name == “methodName”;
```

5 Advices

5.1 Advice Kinds

As pointcuts, we are able to define named advices which can be reused by several different aspects or define an advice directly inside an aspect.

We can define several kinds of advice.

- Code Advice: define chunk of code which can be weaved in methods, constructors, properties
- Type Members Advice: define members (fields, properties, indexers, events, methods, constructors) which can be weaved in classes or structures.
- Interface Members Advice: define members (properties, indexers, events, methods) which can be weaved in interfaces.
- Enum Members Advice: define vales which can be weaved in enumerations.
- Inherited Types Advice: define base type or interfaces which can be weaved in classes or interfaces defining new inheritances
- Types Advice: define types (classes, structures, interface, delegate) which can be weaved in assemblies, classes or structures.
- Attributes Advice: define attribute sections which can be weaved in assemblies, classes or members
- Change Value Advice: define chunk able to change stack values while the IL code is running which can be weaved in methods.

5.2 Code Advice

5.2.1 Explanation

A Code Advice allows to define a chunk of code which can be weaved inside a body method.

5.2.2 Syntax

```
// advice code
advice-code-declaration:
        'advice' 'code' identifier advice-code-block

advice-code-block:
        '{'  [prototype-members-declaration] [statement-list]   '}'

around-statement:
        '[' 'around' 'anchor' ']' '
```

“identifier” is the name of the advice which has to be used in aspects.
Inside a “advice-code-block” you can define:

- ✓ Prototype members which would be explained in a chapter later and used to described the internal member of the target.
- ✓ Instructions which will be weaved in the target

A special instruction has to be used when you want to weave a chunk of code around a join point: **[around anchor];**

5.2.3 Example

In the following example, you will weave

- ✓ Before the target the first System.Console.WriteLine
- ✓ After the target the second System.Console.WriteLine

```
advice code xxxx
{
        System.Console.WriteLine("before");

        [around anchor];

        System.Console.WriteLine("after");
}
```

5.3 Type Members Advice

5.3.1 Explanation

Type members advices allow to defined members that can weave into classes for instances.
You can define named advice or directly define them in the aspect declaration.

Members can be

- ✓ Constant
- ✓ Fields
- ✓ Methods
- ✓ Properties
- ✓ Indexers
- ✓ Events
- ✓ Constructors
- ✓ Destructor
- ✓ Operators

The way to declare most of them is the same as how you will declare them in a C#5 class. Exception has to be done for constructors, destructor and operators. For them, the name of the type is not specified since the members can be woven into any type.

5.3.2 Syntax

```
advice-type-members-declaration:
       'advice' 'type' &'members' identifier advice-type-members-block

advice-type-members-block:
       '{' [prototype-members-declaration] [advice-type-members] '}'

advice-type-members:
       advice-type-member
       advice-type-members advice-type-member

advice-type-member:
       constant-declaration
       field-declaration
       method-declaration
       property-declaration
       event-declaration
       indexer-declaration
       advice-operator-declaration
       advice-constructor-declaration
       advice-destructor-declaration
       advice-static-constructor-declaration

advice-constructor-declaration:
       [attributes] [constructor-modifiers] '(' [formal-parameter-list] ')' [constructor-initializer]
constructor-body

advice-destructor-declaration:
       [attributes] [destructor-modifiers] '~' '(' ')' destructor-body

advice-static-constructor-declaration:
       [attributes] static-constructor-modifiers '(' ')' static-constructor-body

advice-operator-declaration:
       advice-unary-operator-declarator
       advice-binary-operator-declarator
       advice-conversion-operator-declarator

advice-unary-operator-declarator:
       [attributes] operator-modifiers [return-type] 'operator' overloadable-unary-operator '('
advice-operator-declarator-parameter ')' operator-body

advice-operator-declarator-parameter:
       type identifier
       identifier

advice-binary-operator-declarator:
       [attributes] operator-modifiers [return-type] 'operator' overloadable-binary-operator '('
advice-operator-declarator-parameter  ','   advice-operator-declarator-parameter   ')' operator-body

advice-conversion-operator-declarator:
       [attributes] operator-modifiers conversion-operator-type 'operator' [return-type] '(' advice-
operator-declarator-parameter ')' operator-body

conversion-operator-type:
       'explicit'
       'implicit'

advice-operator-declarator-parameter:
       type   identifier
       identifier
```

5.3.3 Example

In the following example, each kind of member has been declared.

```
advice type members myAdvice
{
	// constant
	const int c = 0;

	// event
	event EventHandler sampleEvent;

	// field
	int[] arr;

	// property
	public object Obj {get; set;}

	// event property
	public event EventHandler SampleEvent
	{
		add { sampleEvent += value;}
		remove { sampleEvent -= value;}
	}

	// indexer
	public int this[int i] {get {return arr[i];} }

	// constructor
	(int size) { arr = new int[size]; }

	// method
	[Attribute()]
	public string Method(int a)
	{
		return a.ToString();
	}

	// destructor
	public ~ () { arr = null;}

	// operator unary
	public static operator +(a) { return a; }

}
```

For the operation, as the returned type is not mentioned, **AspectDN** assume that the returned type is the same as the target.

5.4 Types advice

5.4.1 Explanation

Type advices are used to weave new classes, structures, delegates, enumerations or interfaces into assemblies or classes (nested type).

The type that you could be weaved are

- Classes
- Structures
- Interfaces
- Enumerations
- Delegates

Several heterogeneous types can be declared in a same advice.

5.4.2 Syntax

```
advice-types-declaration:
	‘advice’ 'types' identifier '{' [type-declaration*] '}' [';']
```

“identifier” is the name of the advice.

type-declaration is defined is in “ECMA-334 5th Edition / December 2017” as followed

```
type-declaration:
        class-declaration
        struct-declaration
        interface-declaration
        enum-declaration
        delegate-declaration
```

5.4.3 Example

```
advice types myAdvice
{
        // class
        public class BankAccount
        {
                public string Number { get; }
                public string Owner { get; set; }
                public decimal Balance { get; }

                public void MakeDeposit(decimal amount, DateTime date, string note)
                {
                }

                public void MakeWithdrawal(decimal amount, DateTime date, string note)
                {
                }
        }

        // structure
        public struct Coords
        {
                public Coords(double x, double y)
                {
                        X = x;
                        Y = y;
                }

                public double X { get; }
                public double Y { get; }
        }

        // interface
        interface ISampleInterface
        {
                void SampleMethod();
        }

        // enumeration
        enum ErrorCode : ushort
        {
                None = 0,
                Unknown = 1,
                ConnectionLost = 100,
                OutlierReading = 200
        }

        // delegate
        public delegate int PerformCalculation(int x, int y);
}
```

5.5 Interface Members Advice

5.5.1 Explanation

Interface members advices allow to weave interface members in a targeted interface.

Interface Members are the same as you can define in C#5:

- ✓ Properties
- ✓ Methods

- ✓ Indexer
- ✓ Events

5.5.2 Syntax

```
advice-interface-members-declaration:
        advice interface members identifier advice-interface-members-block

advice-interface-members-block:
        { [prototype-members-declaration] [advice-interface-members] }

advice-interface-members:
        interface-member-declaration
        advice-interface-members interface-member-declaration
```

"identifier" is the name of the advice.
You are able to define one or more interface members as is defined in "ECMA-334 5th Edition / December 2017".

5.5.3 Example

```
advice interface members InterfaceMembersAdvice
{
        int P1 {get; set;} // property
        event EventHandler E1; // event
        void Method(int a); // method
        int this[int index] {get; } // indexer
}
```

5.6 Enum members advice

5.6.1 Explanation

These advice weave new enumeration member inside a targeted enumeration.

5.6.2 Syntax

```
advice-enum-members-declaration:
        'advice' 'enum' 'members' identifier enum-body

enum-body:
        '{' [enum-member-declarations] '}'
        '{' enum-member-declarations ',' '}'

enum-modifiers:
        enum-modifier
        enum-modifiers  enum-modifier

enum-modifier:
        'new'
        'internal'
        'protected'
        'private'
        'public'

enum-member-declarations:
        enum-member-declaration
        enum-member-declarations ',' enum-member-declaration

enum-member-declaration:
        [attributes] identifier
        [attributes] identifier '=' constant-expression
```

Identifier is the name of the advice.

Members are described in the same way as you describe them in a C#5 enumeration.

5.6.3 Example

```
advice enum members meAdvice
{
    ConnectionLost1 = 300,
    OutlierReading1 = 400
}
```

5.7 attributes advice

5.7.1 Explanation

These advices allow to weave attribute section towards targeted members or assemblies.

5.7.2 Syntax

```
advice-attributes-declaration:
       'advice' 'attributes' identifier advice-attributes-block

advice-attributes-block:
       '{' [prototype-members-declaration] [attribute-sections] '}'

attribute-sections:
       attribute-section
       attribute-sections attribute-section

attribute-section:
       '['   [attribute-target-specifier]   attribute-list   ']'
       '['   [attribute-target-specifier]   attribute-list   ','   ']'

attribute-target-specifier:
       attribute-target   ':'

attribute-target:
       'field'
       'event'
       'method'
       'param'
       'property'
       'return'
       'type'

attribute-list:
       attribute
       attribute-list   ','   attribute

attribute:
       attribute-name   [attribute-arguments]

attribute-name:
       type-name

+attribute-arguments:
       '('   [positional-argument-list]   ')'
       '('   positional-argument-list   ','   named-argument-list   ')'
       '('   named-argument-list   ')'

positional-argument-list:
       positional-argument
       positional-argument-list   ','   positional-argument

positional-argument:
       [argument-name]   attribute-argument-expression

named-argument-list:
       named-argument
       named-argument-list   ','   named-argument
```

```
named-argument:
        identifier    '='    attribute-argument-expression

attribute-argument-expression:
        ( non-assignment-expression ?! '=' )
```

Identifier is the name of the advice.
Attribute section are described in the same way as you describe them in a C#5.

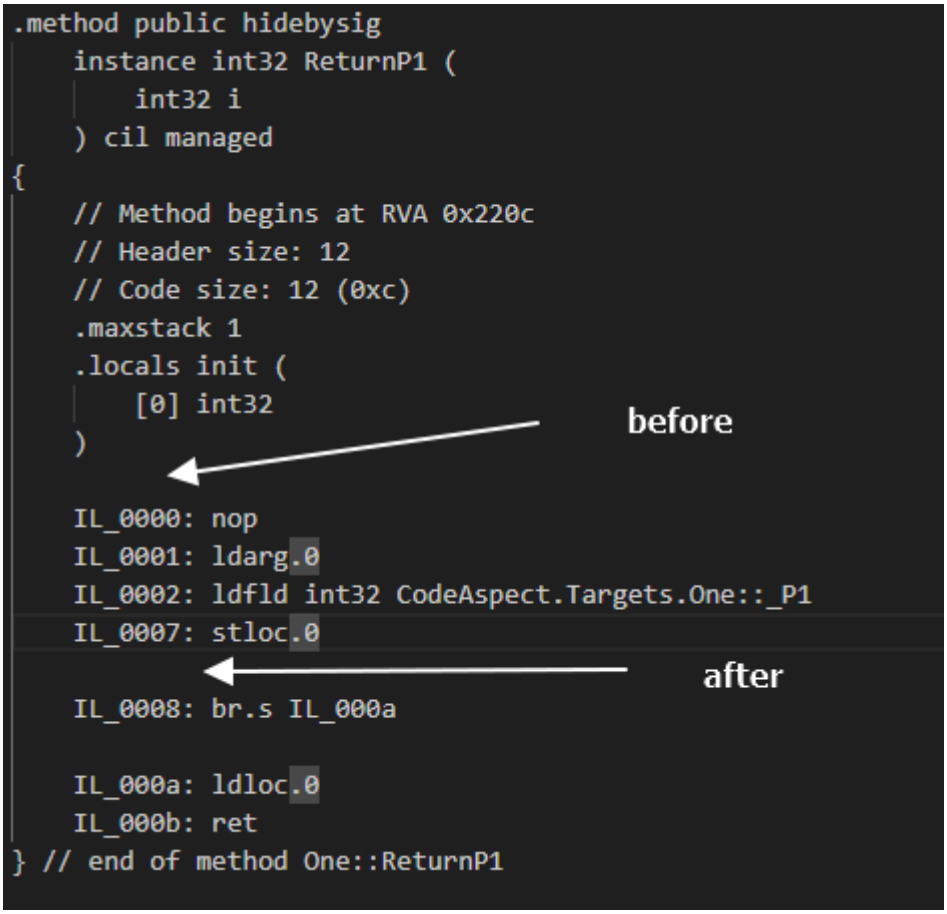
5.7.3 example

advice attributes xxxx
{
    [Att(), Attr()]
    [Att2()]
}

5.8 Change value advice

5.8.1 Explanation

This advice is somewhat special. You have to notice that when an advice is weaved, **AspectDN** is taking in account of the IL stack and preserves it. In the following code, the location where the IL code is weaved if the join point is a get field is as followed.



As you can see, you are not able the value of the field _P1 before the field is saved in the variable 0. Change value advice is used if you want to change the value of the stack after the value _P1 has been stacked.

5.8.2 Syntax

```
advice-change-value-declaration:
       'advice' &'change' 'value' identifier advice-change-value-block

advice-change-value-block:
       type ':'  '{' [prototype-members-declaration] [statement-list]   '}'
```

“identifier” is the name of the advice.
“type” is the type of the value to change.
There is a special keyword to change or get the value of the stack: “value”

5.8.3 Example

In the following example, the value is decreased by 1.

```
advice change value myAdvice
int :
{
       value = value-1;
}
```

6 Prototypes

Prototype are used to describe the context in which the advice has to be targeted and with which they can interact.

AspectDN consider two kinds of prototype:

- ✓ Prototype member which describes internal members of the targeted types.
- ✓ Prototype type which describes the external environment in assemblies in which advice interacts.

6.1 prototypes members

6.1.1 Declaration

6.1.1.1 Explication

Prototypes members are defined in advices and allow to describe members (context) with which advices has to interact.

6.1.1.2 Example:

Imagine that you want track changes of the value of a private field in a class, you need to have a way to look up the value of the concerned field. As we prone that advice has to be independent physically with the target, we need to describe the field as a prototype which will be mapped to the actual field when weaving.

Several kinds of prototype members can be defined

- Field
- Properties and indexes
- Events
- Methods
- constructors
- Type parameters

Parameter type are used when the target (class or method) is generic and the advice need the parameter type to do its job like declare a value with the parameter type.

6.1.1.3 Syntax

```
prototype-members-declaration:
       'prototype' 'members' '{'  [prototype-member-declarations] '}'

prototype-member-declarations:
       prototype-member-declaration
       prototype-member-declarations  prototype-member-declaration

prototype-member-declaration:
       prototype-field-declaration
       prototype-method-declaration
       prototype-indexer-declaration
       prototype-type-parameter-declaration
       prototype-constructor-declaration

prototype-field-declaration:
       [prototype-member-modifier] type   prototype-identifier  ';'

prototype-member-modifier:
       'static'

prototype-method-declaration:
       [prototype-member-modifier] return-type prototype-identifier [type-parameter-list] '('
[formal-parameter-list] ')' ';'

prototype-event-declaration:
       prototype-event-field-declaration

prototype-event-field-declaration:
       [prototype-member-modifier] 'event'   type   prototype-identifier   ';'


prototype-type-parameter-declaration:
       '<'   prototype-identifier '>' ';'

prototype-indexer-declaration:
       [prototype-member-modifier] type '#this' '['   formal-parameter-list ']' '{' prototype-
property-accessor-declarations  '}' [ ';']

prototype-constructor-declaration:
       '#'   '('   [formal-parameter-list]   ')'  ';'

prototype-identifier::
       '#' identifier-or-keyword
       '#'
```

Prototype members are always declared in a special section in the advice

```
prototype members
{
       int #x;
}
```

Prototype member name starts always with the character '#'. The sole modifier allowed is static.
The name of the constructor is only '#' and the indexer is always "this"

Fields are always defined as a field in a class.
Properties and methods are defined as you define them in an interface.

Parameter type are defined in '<' and '>'.

6.1.1.4 Example

```
advice type members myAdvice
{
       prototype members
       {
              <#T>; // type parameter
              T #M<T>(T x); // méthod
              #T #a; // field
              #(#T t); // constructeur
              #this[int i]; // indexer
       }
```

```
        // property advice member using #T, #a
        public #T P2
        {
                get { return #a;}
                set { #a = value;}
        }

        // method using #T, #M, #()
        public #T Get(#T value)
        {
                return #M<#T>(new #(this));
        }
}
```

6.1.2 Prototype Member Mapping

6.1.2.1 Mapping prototype members

6.1.2.1.1 Explanation

If prototype members have been defined in an advice, the aspect will be in charge of defining the mapping of the prototype members with real target members. For instance, a prototype field will be defined with the corresponding field name in the target.

6.1.2.1.2 Syntax

```
prototype-mappings:
        'where'  prototype-mapping-items

prototype-mapping-item:
        prototype-member-mapping
        prototype-type-parameter-mapping
        prototype-type-reference-mapping

prototype-member-mapping:
        prototype-identifier '=' prototype-target-declaration


prototype-target-declaration:
        prototype-target-this-member-declaration
        prototype-target-base-member-declaration
        prototype-target-member-declaration

prototype-target-this-member-declaration:
        'this' '.' identifier

prototype-target-base-member-declaration:
        'base' '.' identifier

prototype-target-member-declaration:
        simple-name
```

Prototype member mapping are always included in “where” clause at the end of an aspect declaration. A prototype member can be mapped to a member target kind according its type.

Prototype member declaration	Prototype member kind	Target member kind	Syntax	Remark
int #f ;	Field	Field	#f = field	The target field must have the same name as the one defined in the left part of the assignment and must have the same return type as the prototype member.
int #f ;	Field	Property	#f = P1	The target property must the same name as name defined in the left part of the assignment and must have the same return type as the prototype member. Moreover, the property must have the correct accessors according the usage in methods.
int #f;	Field	Local variable	#f = 0 or #f = pdb local varname	The corresponding target variable must have the specified index number or have the same name as defined in the left-hand side of the assignment and must have the same return type as the prototype

				member. The name match will only occur if the pdb file is present.
int #f;	Field	Method parameter	#f = param1	The target parameter must have the same name as the one defined in the left part of the assignment and must have the same return type as the prototype member.
int #method(string s) ;	Method	Method	#method = Convert	The target method must have the same name as the one defined in the left part of the assignment and must have the same return type as the prototype member, the same number of method parameter and the same signature.
int #method(string s) ;	Method	Field as delegate	#Method = F1	The target field must have the same name as the one defined in the left part of the assignment and the delegate type of the field must match the method.
event eventhandler #E1;		Event	#E = Event1	The type of the source and the target must match and the name define in the left part of the assignment has to exist in the target type.
int #Pr {get ; set ;}		Property	#Pr = P1	The target property must have the same name as the name defined in the left part of the assignment and must have the same return type as the prototype member. Moreover, the property must have the accessors defined in the prototype member even if one of them is not used.
#(int x) ;		Constructor		Prototype constructors are match impolitely done.
int #this[int a] ;		Indexer		Prototype constructors are match impolitely done.

In C#, we can define new or overloaded methods, we can specify if the mapping has to be done with a member of the target type or on inherited type.

If 'this' is specified (example #f=this. field), the member must be present in the target type (not inherited).
If 'base ' is specified (example #f=base.field), the member must be present in an inherited type of the target type.
If nothing is specified, the first member can be found in the target type or in inherited type (from the hire level to base types).

Remark:
The target member can be an existing member or a member that will be weaved.
The type matching take in account type mapping (advice type or prototype)

6.1.2.2 Prototype member Parameter

6.1.2.2.1 Explanation

In C#, the notion of generic allows the definition of parameter type or method type. In the context of a mapping, it is therefore important to be able to specify this notion and the way to link it with the target.

Imagine that you would weave a new method to a class which is generic. This method would return a value with a type defined in the generic param
On peut imaginer que l'on souhaite rajouter une méthode complémentaire à la classe suivante qui est une classe générique qui réagit en fonction du type de paramètre.

```
/ target class
public class C<t,k>
{
        t P {get; set;}
}

// aspect
existingfields =>
                extend classes : classes.Name == "C`1"
                with type members
                {
                        // prototype member definition
```

```
                        prototype members
            {
                <#x>; // generic parameter
                                #x #Property; // field
            }

                        // new method
            public string GetPropertyValue()
            {
                                string r = null;
                                if (typeof(#x).FullName == typeof(string).FullName)
                                        r = (string)#Property;
                                else
                                        r = #Property.ToSring();
                        }
              }
              where
                        #x = 1, // x is related to the second parameyer of C`1
                        #Property = P; // #Property is related with P of the target

// resulting targen
// target class
public class C<t,k>
{
        t P {get; set;}

        public string GetPropertyValue()
        {
                string r = null;
                if (typeof(k).FullName == typeof(string).FullName)
                        r = (string)P;
                else
                        r = P.ToSring();
        }
}}
```

6.1.2.2.2 Syntax

```
prototype-type-parameter-mapping:
        '<' aspect-identifier '>' ':' prototype-type-generic-parameter-target
        '<' aspect-identifier '>' ':' prototype-method-generic-parameter-target

prototype-type-generic-parameter-target:
        'type' '(' decimal-integer-literal ')'
        'type' '(' type ')'
        decimal-integer-literal
        type

prototype-method-generic-parameter-target:
        'method' '(' decimal-integer-literal ')'
        'method' '(' type ')'
```

You can address the target generic parameter with its index in the target list (starting by 0) or by its name.
If several generic parameters are present and come from both methods and types, the source must be specified (method(0) or type(0))

6.1.2.3 Mapping des advices type

6.1.2.3.1 Explanation

Since **AspectDN** allows types to be woven into assemblies, it is necessary to be able to map them from the target if they are used, as the same type can be woven several times into different namespaces.

6.1.2.3.2 Syntax

prototype-type-reference-mapping:
        unbound-type-name 'from' namespace-or-type-name

Example

```
// type aspect
```

```
 addNewClass =>
       extend assemblies : assemblies.Name == "Targets.dll"
       with types
       {
              public class GenericClass<T> : MarshalByRefObject
              {
                     public T P2 { get; set; }

                     public GenericClass(T a) { P2 = a;}
              }

       }   namespace target PrototypeGenericParameters.New;

// code adspect
codeExtenser =>
       extend before body methods : methods.Name == "SetP3"
       with
       {
              prototype members
              {
                     <#T>;
                     #T #a;
              }

              var mT = new addNewClass.GenericClass<#T>(#a);

              #T result = mT.P2;
              if (typeof(#T).FullName == typeof(#U).FullName)
                     result = (#T)(object)#u;

              return result;

       }
       where <#T> : type(0),#a = A, addNewClass.GenericClass<> from PrototypeGenericParameters.New;
```

6.2 Type of prototypes

6.2.1 Declaration

6.2.1.1 Explanation

AspectDN advocates for weak coupling with the target in order to be able to reuse aspects with several versions of a same application.

We have already addressed the subject on prototype members which allow to describe the target context of a type (internal).

Type prototypes are used to declare the target context within an assembly (external to a type).

The prototype types are classes, structures, interfaces, enumerations and delegates representing roughly a part of the target (the one required for aspects).

Each prototype type has member which are declared in a same way as they were in C#5

6.2.1.2 Syntax

```
prototype-type-declarations:
       prototype-type-declaration
       prototype-type-declarations prototype-type-declaration

prototype-type-declaration:
       prototype-class-declaration
       prototype-interface-declaration
       prototype-struct-declaration
       prototype-delegate-declaration
       prototype-enum-declaration

prototype-class-declaration:
       'prototype' [prototype-class-modifier] 'class' prototype-identifier [type-parameter-list]
[prototype-base-list] [type-parameter-constraints-clauses] prototype-class-or-struct-body
```

```
prototype-class-modifier:
       'abstract'
       'static'

prototype-base-list:
       ':'   prototype-base-types

prototype-base-types:
       prototype-base-type
       prototype-base-types   ','   prototype-base-type

prototype-base-type:
       type

prototype-struct-declaration:
       'prototype' 'struct' prototype-identifier [variant-type-parameter-list] [prototype-base-list]
prototype-class-or-struct-body

+prototype-interface-declaration:
       'prototype' 'interface' prototype-identifier [variant-type-parameter-list] [prototype-base-
list]  interface-body

prototype-delegate-declaration:
       'prototype' 'delegate' return-type prototype-identifier [variant-type-parameter-list] '('
[formal-parameter-list] ')' ';'

+prototype-enum-declaration:
       'prototype' 'enum'  prototype-identifier [enum-base]   enum-body   [';']

prototype-class-or-struct-body:
       '{'   [prototype-type-member-declarations]   '}' [';']

prototype-type-member-declarations:
       prototype-type-member-declaration
       prototype-type-member-declarations prototype-type-member-declarations

prototype-type-member-declaration:
       prototype-type-field-declaration
       prototype-type-property-declaration
       prototype-type-indexer-declaration
       prototype-type-method-declaration
       prototype-type-constructor-declaration
       prototype-type-event-declaration
       prototype-type-nested-declaration

prototype-type-nested-declaration:
       prototype-nested-class-declaration
       prototype-nested-interface-declaration
       prototype-nested-struct-declaration
       prototype-nested-delegate-declaration
       prototype-nested-enum-declaration

prototype-nested-class-declaration:
       'class' identifier [type-parameter-list]  [prototype-base-list]  prototype-class-or-struct-
body

prototype-nested-struct-declaration:
       'struct' identifier [variant-type-parameter-list] [prototype-base-list] prototype-class-or-
struct-body

prototype-nested-interface-declaration:
       'interface' identifier [variant-type-parameter-list] [prototype-base-list]  interface-body

prototype-nested-delegate-declaration:
       'delegate' return-type identifier [variant-type-parameter-list] '(' [formal-parameter-list]
')' ';'

prototype-nested-enum-declaration:
       'enum'  identifier [enum-base]   enum-body   [';']

prototype-type-member-modifier:
       'abstract'
       'static'
       'virtual'

prototype-type-field-declaration:
       [prototype-type-member-modifier] type    identifier  ';'

prototype-type-property-declaration:
       [prototype-type-member-modifier] type    member-name  '{'   prototype-property-accessor-
declarations   '}' [';']
```

```
prototype-property-accessor-declarations:
       prototype-get-accessor-declaration   [prototype-set-accessor-declaration]
       prototype-set-accessor-declaration   [prototype-get-accessor-declaration]

prototype-get-accessor-declaration:
       'get' ';'

prototype-set-accessor-declaration:
       'set' ';'

prototype-type-method-declaration:
       [prototype-type-member-modifier] return-type   identifier   [type-parameter-list] '(' [formal-
parameter-list] ')'  ';'

prototype-type-event-declaration:
       prototype-type-event-field-declaration
       prototype-type-event-property-declaration

prototype-type-event-field-declaration:
       [prototype-type-member-modifier] 'event'   type   identifier   ';'

prototype-type-event-property-declaration:
       [prototype-type-member-modifier] 'event'   type   identifier  '{'   prototype-event-accessor-
declarations   '}'  [';']

prototype-event-accessor-declarations:
       prototype-add-accessor-declaration   prototype-remove-accessor-declaration
       prototype-remove-accessor-declaration   prototype-add-accessor-declaration

prototype-add-accessor-declaration:
       'add' ';'

prototype-remove-accessor-declaration:
       'remove' ';'

prototype-type-indexer-declaration:
       [prototype-type-member-modifier] type    'this'   '['   formal-parameter-list   ']'   '{'
prototype-property-accessor-declarations   '}' [';']

prototype-type-constructor-declaration:
       aspect-identifier  '('   [formal-parameter-list]   ')'  [prototype-type-constructor-
initializer] ';'

prototype-type-constructor-initializer:
       ':'   prototype-type-constructor-initializer-modifier [argument-list]

prototype-type-constructor-initializer-modifier:
       'base'
```

The name of the prototype type must always start with '#'. But the name of the members has to be declarer without '#' and has to have the same name as their counterpart in the target.
All members must be empty (with no implementation as it is when you define interface member)

6.2.2 Mapping

6.2.2.1 Explanation

The mapping will map the prototype type to the target type.

When weaving, the mapping of the members of each prototype type must have the same name and the same characteristics (return type, parameters (generic or method, same accessors).
If this is not respected, weaving errors will be recorded in the log.

6.2.2.2 Syntax

```
prototype-mapping-types-declaration:
       'map' 'types' '{' [prototype-map-type-members] '}' [';']

prototype-map-type-members:
       prototype-map-type-member
       prototype-map-type-members ',' prototype-map-type-member

prototype-map-type-member:
```

```
        prototype-type-name '=' target-type-name //[prototype-mappings-of-types]

prototype-type-name:
        aspect-identifier   [generic-dimension-specifier]
        prototype-type-name  '.'   aspect-identifier   [generic-dimension-specifier]


target-type-name:
        type
```

6.2.3 Example

In the following example, we want to define a price not only for a product but also for the customer/product pair.
We need therefore to weave the customer property into the price class. As this notion does not exist in the definition of aspects, we will declare a prototype of type #customer and this type will be used to declare our property which will be woven into the Price class.

```
// target class
public class Price
{
        Product Product {get; set;}
        decimal Price {get; set;}
}
// target class
public class Customer
{
        int OID {get;}
        string Id {get; set;}
        Address Address {get; set;}
}


// prototype type
prototype class #Customer
{
}

// mapping
 map types
{
                #Customer = Customer
}

// weave new members
// a private field
// a property
// a method
myTyMembersAspect =>
        extend classes : classes.Name = "Price"
        with type members
        (
                @Customer _Customer;
                public #Customer Customer
                {
                        get
                        {
                                return _Customer;
                        }
                        set
                        {
                                _Customer = value;
                        }
                }
        }
```

7 Aspects.

An aspect is defined by different properties.

- A name
- A pointcut referenced by a name or directly defined in the aspect (anonymous)

- An advice referenced by a name or directly defined inside the aspect (anonymous)
- A control flow defining where the advice has to be woven (before, after or around)
- An execution point defining the location in the pointcut. Example call method or body method.
- A mapping clause defining the prototype members mapping.

Different kind of aspects can be defined.

- ✓ Code aspects are used weave chunk of code
- ✓ Type member aspects are used to weave type member toward classes and structures.
- ✓ Interface member aspects are used to weave new interface members toward interfaces.
- ✓ Enum member aspects are used to weave new enumeration member towards enumeration.
- ✓ Inherited type aspects are used to weave base type and interface toward class and structure.
- ✓ Type aspect are used to weave new type (class, delegate, interface, enumeration, structure) toward assemblies or types.
- ✓ Attribute aspect allow to weave new attribute section on class and members.
- ✓ Change value aspect able to change stack value during IL execution.

```
aspect-declarations:
        aspect-declaration
        aspect-declarations aspect-declaration

aspect-declaration:
        aspect-code-declaration
        aspect-type-members-declaration
        aspect-inherit-declaration
        aspect-interface-members-declaration
        aspect-enum-members-declaration
        aspect-types-declaration
        aspect-change-value-declaration
        aspect-attributes-declaration
```

7.1 Control flow et execution time

```
control-flow:
        'set'
        'get'
        'body'
        'call'
        'throw'
        'add'
        'remove'
```

Control flow is only allowed with code advice and defines the location of the pointcut where the chunk of code has to be place.

The possible values are:

- ✓ set: targets an operation of value assignment of a property, field, variable or method argument.
- ✓ set body: targets the body set accessor.
- ✓ get: targets an operation of value retrieving of a to retrieve a value fa property, field, variable or method argument.
- ✓ get body: targets the body get accessor.
- ✓ call: targets an operation of performing a method.
- ✓ body: targets the body of the method.
- ✓ add: targets the event subscription operation
- ✓ add body; targets the body of an event add accessor.
- ✓ remove: targets the removal operation of an event subscription.
- ✓ remove body; targets the body of an event remove accessor.
- ✓ throw: targets the operation that throw of an exception

Controls flows depend on kind of pointcut and can be used only according the list below

	set	set body	get	get body	Call	Body	throw	add	add body	remove	remove body
assemblies											
interfaces											
methods					x	x					
fields	x		x								
properties	x	x	x	x							
events								x	x	x	x
delegates					x						
structs											
exceptions							x				
enum											
constructors					x	x					

Execution time expresses when the advice must be performed.

```
execution-time:
       'before'
       'after'
       'around'
```

- ✓ before: the advice will be executed before instruction or instruction set defined by the pointcut and control flow
- ✓ After: the advice will be executed after the instruction or set of instructions defined by the pointcut and control flow
- ✓ Around: the advice will be executed before and after the instruction or set of instructions. The advice must include the keyword [around anchor]. Instructions preceding the keyword will be executed before the targeted instructions and instructions following the keyword will be executed after the target's instructions.

Remark
With a control flow corresponding to throw, only before mut be used.

The advice weaving is taken in account the stack. In the following method example, if you would weave an advice code before or after the instruction StringReturn ().

```
// CodeAspect.Targets.One
public object OneReturn(string s)
{
    object result = null;
    StringReturn("s");
    return result;
}
```

The IL Code corresponding to the code above is as followed:

```
.method public hidebysig
    instance object OneReturn (
        string s
    ) cil managed
{
    // Method begins at RVA 0x21b4
    // Header size: 12
    // Code size: 21 (0x15)
    .maxstack 2
    .locals init (
        [0] object,
        [1] object
    )

    IL_0000: nop
    IL_0001: ldnull
    IL_0002: stloc.0

    IL_0003: ldarg.0
    IL_0004: ldstr "s"
    IL_0009: call instance string CodeAspect.Targets.One::StringReturn(string)
    IL_000e: pop

    IL_000f: ldloc.0
    IL_0010: stloc.1
    IL_0011: br.s IL_0013

    IL_0013: ldloc.1
    IL_0014: ret
} // end of method One::OneReturn
```

AspectDN does not weave the advice code before or after IL_0009. Instead, the adviceo is placed before IL_0003 or after IL_000e when the stack is empy.

7.2 code aspect

7.2.1 Explanation

Aspect code will weave piece of code contained in the advice to the join point defined in the aspect according to the pointcut, execution time and control flow.

It is possible to use the following pointcut for an aspect code:

- ✓ methods: methods
- ✓ fields: fields
- ✓ properties; property or indexer
- ✓ events; events
- ✓ exceptions: exceptions
- ✓ constructors

7.2.2 Syntax

```
aspect-code-declaration:
       identifier '=>' 'extend' &execution-time control-flows aspect-pointcut 'with' aspect-advice-
code-named    [prototype-mappings]  ';'
       identifier '=>' 'extend' &execution-time control-flows aspect-pointcut 'with' aspect-advice-
code-anonymous  [';']
       identifier '=>' 'extend' &execution-time control-flows aspect-pointcut 'with' aspect-advice-
code-anonymous  prototype-mappings ';'

aspect-advice-code-named:
       qualified-identifier
```

```
aspect-advice-code-anonymous:
       '{' [prototype-members-declaration] [statement-list]   '}'
```

"identifier" is the name of the aspect which will be used to reference the aspect and call back in error logs if any anomalies were found.

The pointcut can be a named pointcut (see dedicated chapter) or directly declared in the aspect. In this case, we talk about anonymous pointcut since it has no name.

```
aspect-pointcut-anonymous:
       aspect-pointcut-common-anonymous
       aspect-pointcut-this-code-anonymous

aspect-pointcut-common-anonymous:
       pointcut-type ':' pointcut-expression

aspect-pointcut-this-code-anonymous:
        pointcut-type ':' pointcut-expression 'in' 'prototype' 'type' namespace-or-type-name
```

Example:

```
methods xxx : methods.Name == “methodName”;
```

corresponds to the following declaration in an aspect

```
methods: methods.Name == “methodName”;
```

The pointcut "this" is special and can only be defined in an anonymous pointcut within an aspect. The declared type in the anonymous pointcut must be a prototype type.
When the advice related to the aspect is weaving, the word 'this' or any code in the aspect's advice code will be able to access all the members of the prototype type defined within it initially or by another aspect with a pointcut of type "this" on the same prototype type.

7.2.3 Example

```
// prototype declaration
prototype class #Pr
{
       public int P1 {get; set;}
}

// aspect code declaration with this pointcut
myCodeAspect =>
       extend before set body properties : properties.Name="P1" in prototype type #Pr
       with
       {
              System.Console.WriteLine(string.format("value before set P1 from the class {0} : {1}",
typeof(#Pr).ToString(), P1 ));
       }
```

In the above example, we describe a prototype type #Pr in which a property P1 is declared.
The declared advice code can use the property of the type prototype because we have a pointcut of type "this".
The instructions in the advice code will be woven before the assignment of the value to the P1 property is carried out and this for all the target classes correctly mapped to the type prototype.

7.3 type members aspect

7.3.1 Explanation

Aspect codes will allow to weave new members of an advice to the join point defined in the aspect.

The two different pointcuts usable are:

- ✓ classes: classes
- ✓ structs: structures

The members which available are described in the chapter “type members advice”. Of course, prototype member et prototype are allowed in all the declaration and the mapping has to be done.

7.3.2 Syntax

```
aspect-type-members-declaration:
	identifier '=>' 'extend' aspect-pointcut 'with' 'type' 'members' aspect-advice-type-members-
named [aspect-type-member-modifiers-declaration] [prototype-mappings]  ';'
	identifier '=>' 'extend' aspect-pointcut 'with' 'type' 'members' aspect-advice-type-members-
anonymous [';']
	identifier '=>' 'extend' aspect-pointcut 'with' 'type' 'members' aspect-advice-type-members-
anonymous aspect-type-member-modifiers-declaration [prototype-mappings]  ';'
	identifier '=>' 'extend' aspect-pointcut 'with' 'type' 'members' aspect-advice-type-members-
anonymous prototype-mappings  ';'
```

« identifier » is the name of the aspect.
Pointcut can be a named pointcut or an anonymous one.

```
aspect-pointcut-anonymous:
	aspect-pointcut-common-anonymous
	aspect-pointcut-this-type-members-anonymous

aspect-pointcut-common-anonymous:
	pointcut-type ':' pointcut-expression

aspect-pointcut-this-type-members-anonymous:
	'prototype' 'type' namespace-or-type-name
```

Modifiers allow to specify, when a new member is weaving, whether the member is new or must override an existing member. This modifier is applied to all members of the advice contained in the aspect and apply only if the member is already in the target.

```
aspect-type-member-modifiers-declaration:
	'modifiers' ':' aspect-type-member-modifiers

aspect-type-member-modifiers:
	aspect-type-member-modifier
	aspect-type-member-modifiers ',' aspect-type-member-modifier

aspect-type-member-modifier:
	'new'
	'override'
```

7.3.3 Example

In the following example, AspectDN will implement a method ToString which will override the existing one. This woven method will use information from the target class namely Id and CompanyName.

```
addCustomerAspect =>
	extend classes : classes.FullName == "Sales.Customer"
	with type members
	{
		prototype members
		{
			string #Id;
			string #CompanyName;

		}


		public string ToString()
```

```
            {
                    return String.Format("Customer {0}, {1}", #Id, #CompanyName);
            }
        } modifiers : override
        where #Id = Id, #CompanyName = CompanyName;
```

7.4 type aspect

7.4.1 Explanation

Type Aspect allow to weave types defined in an advice to the pointcut defined or identified in the aspect.

The possible pointcuts are:

- ✓ assemblies: allows to define a module in the assembly
- ✓ classes: classes
- ✓ structs: structures

7.4.2 Syntax

```
aspect-types-declaration:
        identifier '=>' 'extend' aspect-pointcut 'with' 'types' aspect-advice-type-named  aspect-type-
target-namespace [prototype-mappings-of-types] ';'
        identifier '=>' 'extend' aspect-pointcut 'with' 'types' aspect-advice-type-named [prototype-
mappings-of-types] ';'
        identifier '=>' 'extend' aspect-pointcut 'with' 'types' aspect-advice-type-anonymous aspect-
type-target-namespace [prototype-mappings-of-types] ';'
        identifier '=>' 'extend' aspect-pointcut 'with' 'types' aspect-advice-type-anonymous  [';']
        identifier '=>' 'extend' aspect-pointcut 'with' 'types' aspect-advice-type-anonymous
prototype-mappings-of-types ';'

aspect-type-target-namespace:
         'namespace' 'target' qualified-identifier

prototype-mappings-of-types:
        'where' prototype-mapping-types

prototype-mapping-types:
        prototype-mapping-type
        prototype-mapping-types ',' prototype-mapping-type

prototype-mapping-type:
        prototype-type-parameter-mapping
        prototype-type-reference-mapping

prototype-type-parameter-mapping:
        '<' aspect-identifier '>' ':' prototype-type-generic-parameter-target
        '<' aspect-identifier '>' ':' prototype-method-generic-parameter-target

prototype-type-generic-parameter-target:
        'type' '(' decimal-integer-literal ')'
        'type' '(' type ')'
        decimal-integer-literal
        type

prototype-type-reference-mapping:
        unbound-type-name 'from' namespace-or-type-name
```

The namespace is the name of the target namespace if the type is woven into an assembly. If the target is a class or structure, the type will be woven as a nested type.

Here, prototype members mapping is slightly different as you can only define generic parameters as prototype members.

7.4.3 Example

In the following example, in the SalesDb.dll assembly, two new types will be woven: a class and an interface. Note that the class inherits from an interface defined in the same aspect. The interface could have been defined in another advice. In the latter case, the interface would have been addressed with the package name

of the advice, the name of the advice and the name of the interface. In addition, you would have had to map the interface from the right namespace.

The class and the interface will be woven in the same namespace: SalesDb

.

```
myAspect =>
	extend assemblies : assemblies.Name == "SalesDb.dll"
	with types
	{
		internal class Publisher : IPublisher
		{
			internal string Id { get; set; }
			internal string PublisherName { get; set; }

			string IPublisher.Id { get { return Id; }}
			string IPublisher.PublisherName { get { return PublisherName;} set
{PublisherName = value;} }
		}

		public interface IPublisher
		{
			string Id { get; }
			string PublisherName { get; set; }
		}
	} namespace target SalesDb;
```

7.5 interface members aspect

7.5.1 Explanation

Interface members aspect allow to weave members defined in an advice to one or more target interfaces.

The possible pointcut is:

- ✓ interfaces: interfaces

Important note:
An interface member can only be woven to an interface if all types implementing that interface have that member. This member can be already existing or a member being woven in another advice. Otherwise, an error message will indicate that weaving cannot be performed.

7.5.2 Syntax

```
aspect-interface-members-declaration:
	identifier '=>' 'extend' aspect-pointcut 'with' &'interface' 'members' aspect-advice-
interface-members-named  [prototype-mappings] ';'
	identifier '=>' 'extend' aspect-pointcut 'with' &'interface' 'members' aspect-advice-
interface-members-anonymous [';']
	identifier '=>' 'extend' aspect-pointcut 'with' &'interface' 'members' aspect-advice-
interface-members-anonymous prototype-mappings';'
```

7.5.3 Example

In the following example, the target class CustomerOrder implements an ICustomerOrder interface.
We weave a new interface member named NetAmount to the ICustomerOrder interface without forgetting to weave the NetAmount member in the CustomerOrder class which does not already exist otherwise a weaving error would be detected.

```
addICustomerOrderMembers =>
	extend interfaces : inerfaces.Name = "ICustomerOrder"
	with interface members
	{
		decimal NetAmount { get;}
	}

addCustomerOrderMembers =>
```

```
        extend classes : classes.Name = "CustomerOrder"
        with type members
        {
                internal decimal NetAmount
                {
                        get { return CustomerOrderLines.Sum(t => t.Amount * (1 - (t.DiscountRate() /
100)));}
                }
        }
```

7.6 enum members aspect

7.6.1 Explanation

Enum members aspect allow to weave enum members defined in an advice to one or more target enumeration.

Possible pointcut is

- ✓ enums : enumeration

7.6.2 Syntax

```
aspect-enum-members-declaration:
        identifier '=>' 'extend' aspect-pointcut 'with' &'enum' 'members' aspect-advice-enum-members-
named   ';'
        identifier '=>' 'extend' aspect-pointcut 'with' &'enum' 'members' aspect-advice-enum-members-
anonymous [';']
```

7.6.3 Example

In the following example, two new members are woven to target enumeration named ‘Kinds’.

```
myAspect =>
        extend enums : enums.Name = "Kinds"
        with enum members
        {
                B = 4,
                C = 8
        }
```

7.7 attribute aspect

7.7.1 Explanation

Attribute aspects allow to weave attributes to one or more targets according its defined pointcuts.
assemblies: assembly or module

- ✓ classes: classes
- ✓ interfaces: interfaces
- ✓ methods
- ✓ fields
- ✓ properties; property or indexer
- ✓ structs: structures
- ✓ constructors

7.7.2 Syntax

```
aspect-attributes-declaration:
        identifier '=>' 'extend' aspect-pointcut 'with' &'attributes' aspect-advice-attributes-named
[prototype-mappings]  ';'
        identifier '=>' 'extend' aspect-pointcut 'with' &'attributes' aspect-advice-attributes-
anonymous [';']
```

```
        identifier '=>' 'extend' aspect-pointcut 'with' &'attributes' aspect-advice-attributes-
anonymous  prototype-mappings  ';'
```

7.7.3 Example

In the following example, we weave two new attribute types to the target.dll assembly. From these two new attribute types, we weave attribute sections to the target Class named "TargetClass"

```
typesAspect =>
        extend assemblies : assemblies.Name=="target.dll";
               with types
               {
                       public class Attribute1 : Attribute
                       {
                       }

                       public class Attribute2 : Attribute
                       {
                               public string Id { get; set;}

                               public Attribute2(string id) { Id = id; }
                       }
    } namespace target AttributesAspect.Targets;

attributesAspect =>
        extend classes : classes.Name=="TargetClass"; with attributes
        {
                [AttributeTypesAdvice.Attribute1()]
                [AttributeTypesAdvice.Attribute2("xxx")]
        }
 where AttributeTypesAdvice.Attribute1 from AttributesAspect.Targets, AttributeTypesAdvice.Attribute2
from AttributesAspect.Targets;
```

7.8 change value aspect

7.8.1 Explanation

Change value aspect allow to weave code which is able to modify a stack value towards the join point defined in the aspect according to the pointcut, the execution time and the control flow.

As a reminder, **AspectDN** weave code before or after a set of IL Code with a stack counter equal to 0 (see change value advice for more explanation).

It is possible to use the following pointcut:

- ✓ methods: methods
- ✓ fields: fields
- ✓ properties; property or indexer
- ✓ constructors: constructors

7.8.2 Syntax

```
aspect-change-value-declaration:
        identifier '=>' 'change' 'value' 'when' control-flows aspect-pointcut 'with' aspect-advice-
changevalue-named     [prototype-mappings]  ';'
        identifier '=>' 'change' 'value' 'when' control-flows aspect-pointcut 'with' aspect-advice-
changevalue-anonymous  [';']
        identifier '=>' 'change' 'value' 'when' control-flows aspect-pointcut 'with' aspect-advice-
changevalue-anonymous  [prototype-mappings]  ';'
```

7.8.3 Example

In the following example, at each time the property named P1 is initialised with a value, the value will be stored by divided it by two.

```
myaspect =>
        change value when set properties  : properties.Name == "P1"
```

```
        with
        int :
        {
                value = value/2;
        }
```

7.9 inherited aspect

7.9.1 Explanation

Inherited aspects allow you to weave types as base types or interfaces if the target implements the interface members.

It is possible to use the following pointcut:

- ✓ classes: classes
- ✓ interfaces: interfaces
- ✓ structs: structures

When constructors are existing in the target which will have a new base type, you are able to constructor overloading by matching the signature of the inherited constructor with the constructor of the targeted type.

7.9.2 Syntax

```
aspect-inherit-declaration:
        identifier '=>'  aspect-pointcut &'inherit' 'from' aspect-advice-inherit-anonymous [override-
constructor-declaration-section] [prototype-mappings] ';'* override-constructor-declaration-section:
        'override' 'constructors' override-specific-constructor-declarations

override-specific-constructor-declarations:
        override-specific-constructor-declaration
        override-specific-constructor-declarations ',' override-specific-constructor-declaration

override-specific-constructor-declaration:
        override-specific-constructors ':'  [argument-list]

override-specific-constructors:
        override-specific-constructor
        override-specific-constructors ',' override-specific-constructor

override-specific-constructor:
        '('   [formal-parameter-list]  ')'
```

The constructor overload is mandatory for constructors. The left-hand side describes the constructor(s) of the target type and the right-hand side the constructor of the base type.
Constants or expressions can be used in the right-hand part.

7.9.3 Example

In the following example, the weaver adds a basic type to the outer class. The two constructors of class "Outer" overload the base constructor of the class "A" with the parameter values a and b.

```
inheritAspect =>
        classes : classes.Name="Outer"
        inherit from A, I1, I2
        override constructors
             (int a, string b, object o), (int a, string b, EventHandler c)  : (a , b);
```

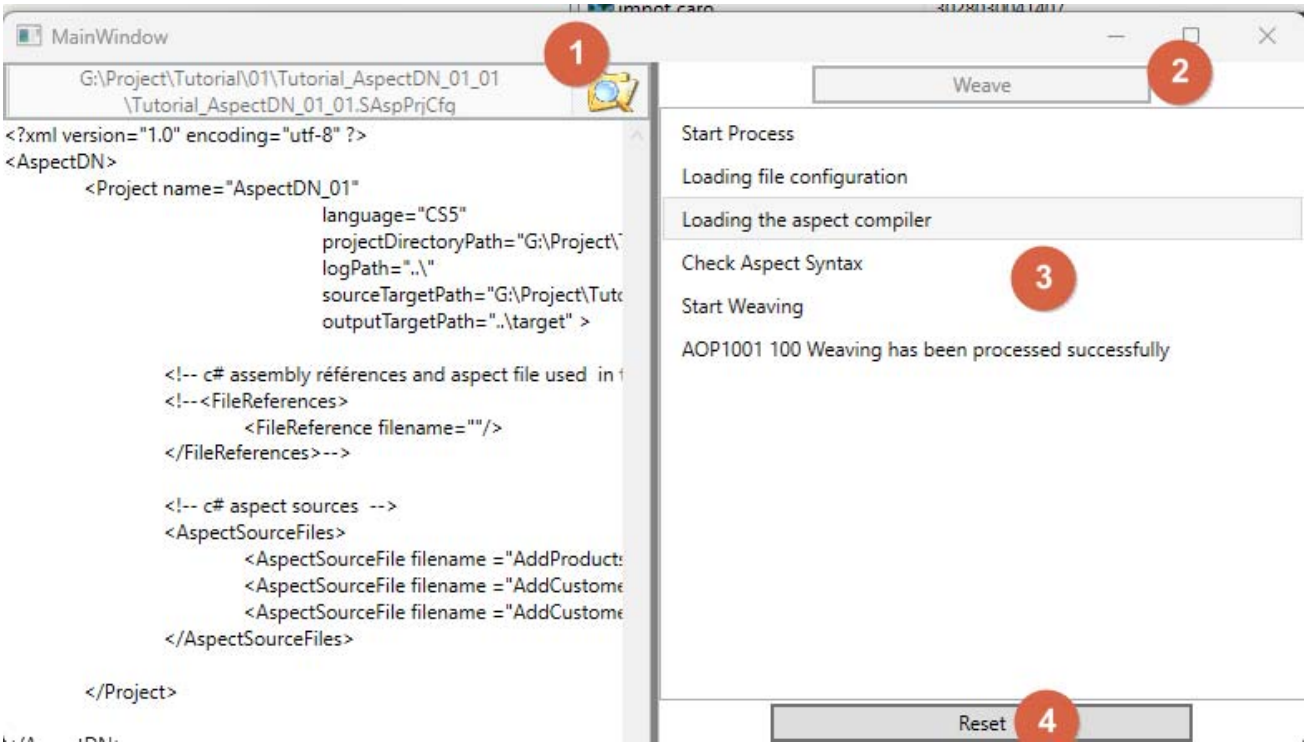
8 Launch a weave process

8.1 Weave operation

Weaving command line

The weaving is done by running the following command: aspectdn -create "project-name"
aspectdn is the executable in charge of weaving an aspect project.
-create is the parameter expressing that we want create new executables from the target by weaving aspects.

Alternatively, you can launch the command: aspectdn -window.



With the windows, you can:

(1) Select the aspectdn project file
(2) Weave aspects
(3) See the process
(4) Reset for a new weaving or close the window

The project is an xml file that you have to declare which contains all the information to perform the weaving (file containing aspect, target assemblies, reference assemblies to add to the target and the location of the woven assemblies)

The weaving process proceed as followed.

- ✓ It performs a syntactic analysis of the prototypes, advices, points and aspects.
- ✓ It visits the target assembly to locate all join point.
- ✓ According the join point define by the aspect, AspectDN weave the advice to the target and save the new executable file in a directory defined by the project.
- ✓ In case of errors has been encountered, either syntactic errors or integrity errors, a log is then updated indicating the anomalies. According the error, the woven assemblies are not created.

8.2 AspectDN Project

Weaving project are saved in an XML file with the following structure:

```
<?xml version="1.0" encoding="utf-8" ?>
<AspectDN>
        <Project name=""
                language=""
                projectDirectoryPath=""
```

```
                logPath=""
                sourceTargetPath=""
                outputTargetPath="">

                <!-- c# assembly and directories excluded for this project -->
                <SourceTargetExclusion>
                        <File filename=""/>
                </SourceTargetExclusion>

                <!-- c# assembly références and aspect file used  in this project -->
                <FileReferences>
                        <FileReference filename=""/>
                </FileReferences>

                <!-- c# aspect sources  -->
                <AspectSourceFiles>
                        <AspectSourceFile filename =""/>
                </AspectSourceFiles>

        </Project>
</AspectDN>
```

Attribute "name" is the prject name used to identify the project.

Attribute "projectDirectoryPath" define the project directory where the **AspectDN** files declaring the aspects, pointcuts and advices are located. Only one directory can be defined.

Note
For all other attributes containing a directory name, if they are started with '..\', the system will complete the directory path with the project directory path.

Attribute "logPath" allows to specify the directory where the log file of anomalies will be generated.

Attribute "sourceTargetPath" is the directory where the assemblies for which we want to apply a weaving.

Attribute "outputTargetPath" is the target directory where woven assemblies are stored.

The SourceTargetExclusion section allow to exclude some assemblies present in the source target path but not targeted by the aspect and therefore which can be excluded in the weaving process. This has a big impact on performance as we can limited a huge amount of pointcuts.

Note you cannot address the assemblies of the .net framework with **AspectDN**.

In the FileReferences section, you specify the files necessary to carry out the weaving and the coherence of the assemblies after the weaving as **AspectDN** consider all assemblies as a coherent whole (references must exist). This could be a dll such as log4net.dll but also a pre-compiled aspect file (suffixed with aspectdn).

AspectSourceFiles is the section where you indicate the set of sources containing the declarations of aspects, advices, pointcuts and prototypes.