What is new in Eos 0.2?

1. Introductions:

An introduction represents static crosscutting. The syntax of an introduction in Eos is as follows:

```
introduce in TypePattern
{
 // Members to be introduced.
}
```

We decided to use a different syntax from AspectJ for two reasons:

1) This syntax is more readable. AspectJ syntax for introduction is conflicting with the syntax of a code type member implementing an interface type member, so it is not clear by just looking at the code whether a given type member is implementing an interface type member or introducing an type member into other type. We have decided to make it more explicit by saying explicitly introduce following elements in the types that match the TypePattern.

2. Code members being introduced to a given set of types can now be grouped into a block thus eliminating the duplication of TypePattern.

An introduction can introduce the following type of members to a class or an aspect. An introduction may not introduce any new members in an interface.

Constant
Field
Method
Property
Event
Indexer
Operator
Constructor
Destructor
Internal classes

In addition to the language constructs above introductions can also introduce the following crosscutting members into another aspects.

Named pointcut
Advice
role
action

2. Around Advices:

Around advices can advice execution, field get, field set, and instance constructor execution/object initialization join points. An around advice may not advice an exception handler join point. Syntax of an around advice is as follows:

Return Type around(Formal Parameters) : Pointcut { Body }

It is a compile time error to advice a join point with a non-void return type with an around advice returning void. It is a run-time error (System.InvalidCastException) to return null or objects in the around advice that may not be casted to the return type of the join point.

Join Points	**Return Type**
Method execution	Return type of the method.
Field Set	Type of the field
Field Get	Void
Constructor execution	Void

3. New join points:

Exception handlers: When exception handling code (catch clauses) executes in a program. The argument of the catch clause (the exception being caught) is considered as the argument available at the join point and it can be accessed inside the advice using thisJoinPoint.getArgs(). No value is returned from a exception handler join point, so its return type is considered to be void. The expression thisJoinPont.getReturnValue() thus returns "null" for this join point.

Constructor execution: When an instance constructor actually executes after the chained base (:base(..)) and this constructors (:this(..)). The object being constructed is the currently executing

object, and so may be accessed with the expression thisJoinPoint.getThis(). No value is returned from a instance constructor execution join point, so its return type is considered to be void. The expression thisJoinPont.getReturnValue() thus returns "null" for this join point.

Type constructor execution: When the type constructor for a class executes. No value is returned from a type constructor execution join point, so its return type is considered to be void. The expression thisJoinPont.getReturnValue() thus returns "null" for this join point.

Property get execution: When the get accessor for a property executes. The value returned from this join point is of the same type as the property.

Property set execution: When the set accessor for a property executes. The value returned from this join point is considered to be void. The variable bound to "value" is considered as the argument available at the join point and it can be accessed inside the advice using thisJoinPoint.getArgs().

4. New pointcut designators:

handler(type pattern): Selects exception handler join points where the type of the exception being caught matches the type pattern.

initialization(constructor pattern): Selects constructor execution join points, where the signature of the join points matches the constructor pattern.

staticinitialization(type pattern): Selects the type constructor of each type whose signature matches type pattern.

within(type pattern): Selects join points that are defined in the type that matches type pattern.

withincode(method pattern): Selects join points that are defined in all methods whose signature matches the method pattern.