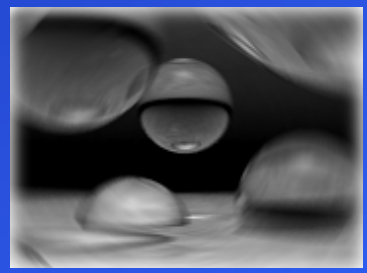
ISaI Ingeniería del Software y Sistemas de Información

Tecnologías Orientadas a Aspectos en .NET



Cristóbal Costa © Abril 2005

Aproximaciones AOP

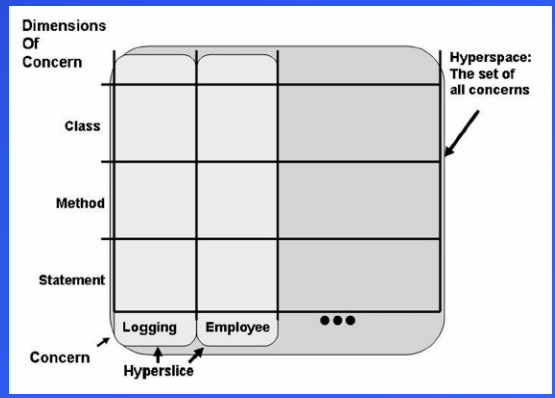
- ✓ AspectJ y Hyper/J (Java)
- ✓ CLAW y AOP# (Primeras aproximaciones para .NET)
- ✓ AOP.NET (Proyecto Siemens, modificación del código en ejecución)
- ✓ AOPdotNetAddin
- ✓ AspectC#
- ✓ JAsCo.NET
- ✓ Loom.NET y Rapier-Loom.NET
- ✓ SetPoint!
- • **ASPECT.NET** (Prototipo de Microsoft)
- ✓ EOS
- • Weave.NET
  - – (Grupo de AspectC#, AOSD Case Studies, SourceWeave.NET, Theme)
- • Otros:
  - – AspectDNG, DotNetAOP, AOPNet, ContextBoundModel, Spring.NET (muy recientes)

Precedentes

- **AspectJ** (http://eclipse.org/aspectj/)
  - A partir del código fuente y las definiciones de aspectos, realiza el ***Weaving*** o "entretejido" de código entre ambos
  - **JoinPoint:** Instante de la ejecución de un programa susceptible de ser interceptado por un aspecto
  - **PointCut**: Define un conjunto de *joinpoints* a los cuales se les aplicará un aspecto
  - **Advice:** Define el código que se ejecutará en los *joinpoints* y cómo se debe combinar (*After*, *Before* o *Around*).
  - **Aspect:** Es la estructura sintáctica que encapsula varios *crosscutting concerns*.

Precedentes

- **Hyper/J** (http://www.research.ibm.com/hyperspace/index.htm/)
  - Enfoca los aspectos en dimensiones, de forma que cada dimensión se encarga de un *concern* concreto
  - HyperSpace: Es el conjunto de todas las unidades susceptibles de ser combinadas (paquetes, clases, interfaces, …)
  - HyperSlice: Conjunto de unidades declarativamente completas que definen un *concern*
  - HyperModules: Composición de HyperSlices



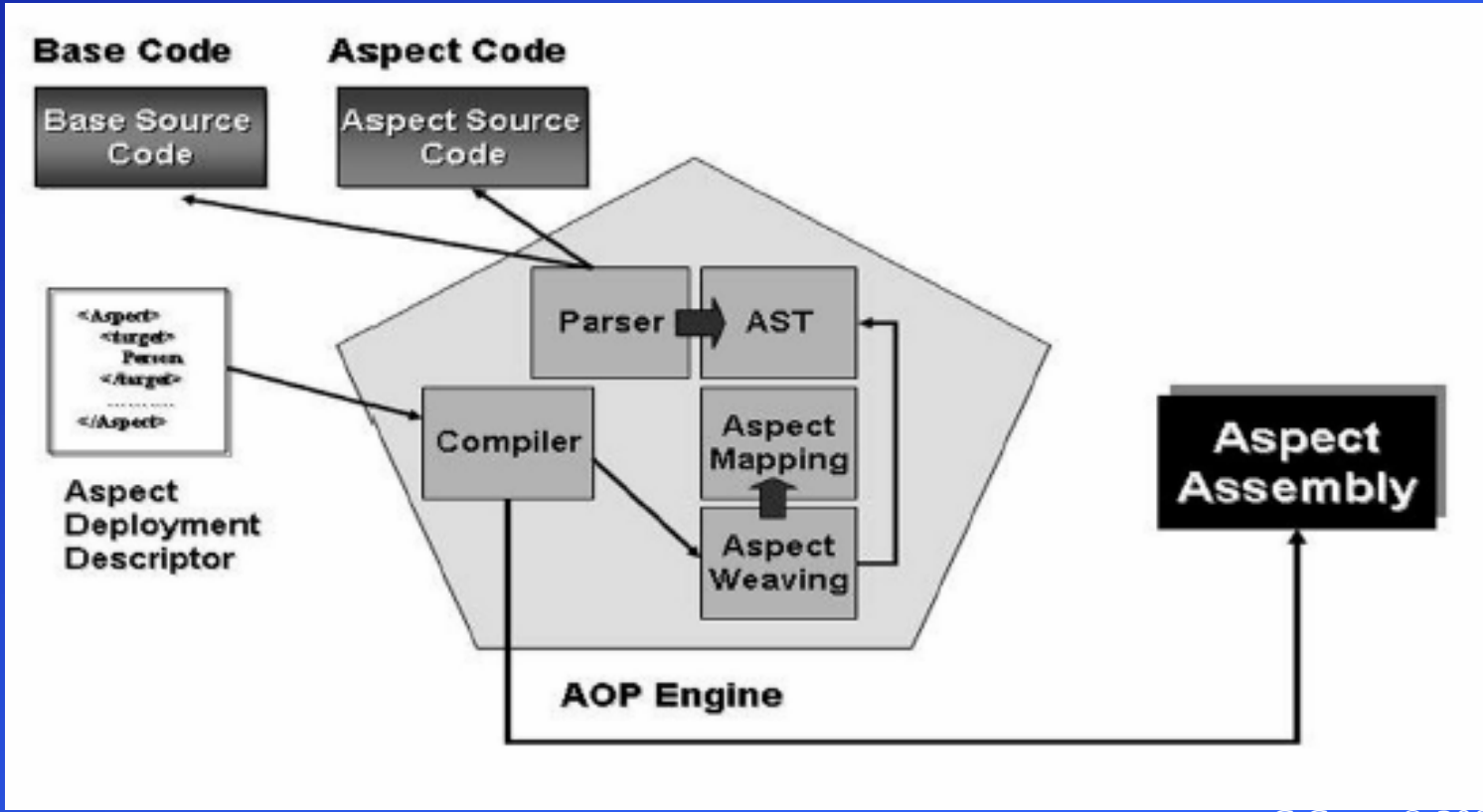
AspectC#

http://www.dsg.cs.tcd.ie/index.php?category_id=169

- **Funcionalidad aportada:**
  - Separa el código base y los aspectos
  - No extiende el lenguaje
- **Inconvenientes:**
  - Prototipo no actualizado desde 2002
  - El *parser* no admite anidaciones de clases, estructuras, delegados, eventos, atributos y expresiones en métodos en el código de los aspectos
  - No ha sido probado en aplicaciones de envergadura
  - Los aspectos se añaden en tiempo de **compilación**

AspectC# - Compilador

- Se compila el codigo y se genera un nuevo fichero .CS
- Problema: se limita la expresividad original del código sino se implementa todo, como p.ej. la situación de las llaves



AspectC# - Ejemplo (I)

Código BASE

```
using System;

namespace Test
{
    // Ejemplo de código Base
    public class Base
    {
        public Base()
        {
            // TODO: Add constructor logic here
        }

        public void Saluda()
        {
            Console.WriteLine("Hola Mundo");
        }

        public int funcion(int a, int b)
        {
            return a+b;
        }

        public static void Main()
        {
            Base hm = new Base();
            hm.Saluda();
            Console.WriteLine("Sumando 4+5: " + hm.funcion(5,4));
            string s = Console.ReadLine();
        }
    }
}
```

AspectC# - Ejemplo (I)

Código BASE

```
using System;

namespace Test
{
    //namespace OtroNameSpace
    pu{
    {
        /// Este es el código de un Aspecto cualquiera
        /// NOTAS:
        ///     - El compilador es muy sensible. No soporta muchas cosas (y no están documentadas),
        ///       como situar llaves en la misma linea que la declaracion,
        public abstract class AspectoFuncional
        {
            // Ejemplos de propiedades de clase
            private string propiedad1 = "propiedad";
            public int propiedad_int = 2;

            public AspectoFuncional()
            {
                // Aqui el constructor del codigo
            }

            public void EmitirMensaje()
            {
                Console.WriteLine("Soy el metodo EmitirMensaje de la clase AspectoFuncional");
                Console.WriteLine("Estoy introducido en el metodo: " + ThisJoinPoint.GS_MethodName);
                Console.WriteLine("");
            }
    }
}
```

Código de un Aspecto

AspectC# - Ejemplo (I)

```
using System;

namespace Test
{
    //namespace OtroNameSp
    pu{
    {
        /// Este es el c
        /// NOTAS:
        ///     - El com
        ///       como s
        public abstract
        {
            // Ejemplos
            private stri
            public int p

            public Aspect
            {
                // Aqui
            }

            public void
            {
                Console.
                Console.
                Console.
            }
    }
}
```

Aspect Descriptor

```
<?xml version="1.0" encoding="utf-8" ?>
<Aspect>
    <TargetBase>C:\AspectCSDemo\CodigoBase</TargetBase>
    <AspectBase>C:\AspectCSDemo\CodigoAspectos</AspectBase>
    <Aspect-Method>
        <Name>AspectoFuncional1</Name>
        <Namespace>OtroNameSpace</Namespace>
        <Class>AspectoFuncional</Class>
        <Method>EmitirMensaje()</Method>
    </Aspect-Method>
    <Aspect-Method>
        <Name>AspectoFuncional2</Name>
        <Namespace>OtroNameSpace</Namespace>
        <Class>AspectoFuncional</Class>
        <Method>Suma(int a, int b)</Method>
    </Aspect-Method>
    <Target>
        <Namespace>Test</Namespace>
        <Class>Base</Class>
        <Method>
            <Name>Saluda()</Name>
            <Type>after</Type>
            <Aspect-Name>AspectoFuncional1</Aspect-Name>
        </Method>
        <Method>
            <Name>Saluda()</Name>
            <Type>before</Type>
            <Aspect-Name>AspectoFuncional1</Aspect-Name>
        </Method>
```

```
spectо

documentadas),

cional");
GS_MethodName);
```

AspectC# - Ejemplo (II)

Fichero resultante

```
namespace Test {
    using Ie.Tcd.AspectCSharp;
    using System;

    public class Base {
        private string test;

        public Base() {
            JoinPoint ThisJoinPoint = new JoinPoint("Base ","Base",@"C:\AspectCSDemo\CodigoBase\Base.c
            // TODO: Add constructor logic here
        }

        public virtual void Saluda() {
            JoinPoint ThisJoinPoint = new JoinPoint("Base ","Saluda",@"C:\AspectCSDemo\CodigoBase\Base

            Console.WriteLine("Soy el metodo EmitirMensaje de la clase AspectoFuncional");
            Console.WriteLine("Estoy introducido en el metodo: " + ThisJoinPoint.GS_MethodName);
            Console.WriteLine("");

            Console.WriteLine("Hola Mundo");

            Console.WriteLine("Soy el metodo EmitirMensaje de la clase AspectoFuncional");
            Console.WriteLine("Estoy introducido en el metodo: " + ThisJoinPoint.GS_MethodName);
            Console.WriteLine("");
        }

        // Method renamed to: funcion001a
        public virtual int funcion(int a, int b) {

            if(a+b > 10)
            {
```

JAsCo.NET

http://ssel.vub.ac.be/jasco/jascodotnet.php

- **Funcionalidad aportada:**
  - Extiende el lenguaje con ASPECTOS y CONECTORES
  - Orientado a los **Componentes**
  - Los aspectos son reusables en tiempo de ejecución y pueden ser incorporados dinámicamente en el sistema
  - Los aspectos se pueden combinar entre sí
  - No afecta a los COTS

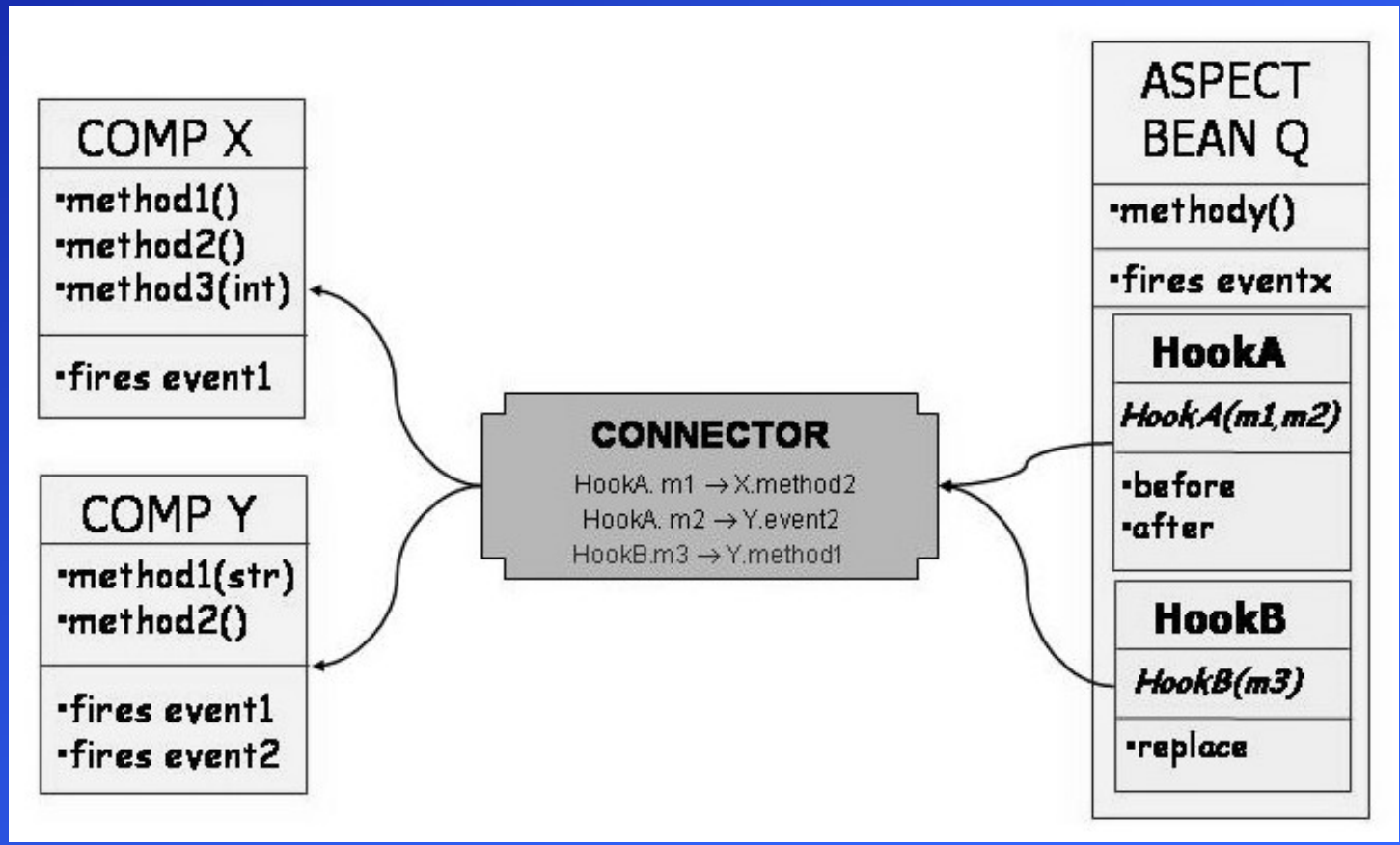
  - Los Aspectos proporcionan:
    - Agrupación de *Concerns* **(Hooks)**
    - Cuándo y Cómo se activarán *(before, after o replace)*
    - Permiten **Herencia** de aspectos, **Abstracción**, **Transiciones de estados**

JAsCo.NET

- **Funcionalidad aportada (II):**
  - Los Conectores proporcionan:
    - Enlace con métodos mediante comodines
    - Enlace con métodos, instancias, o clases
    - Definición de relaciones de precedencia y encadenamientos

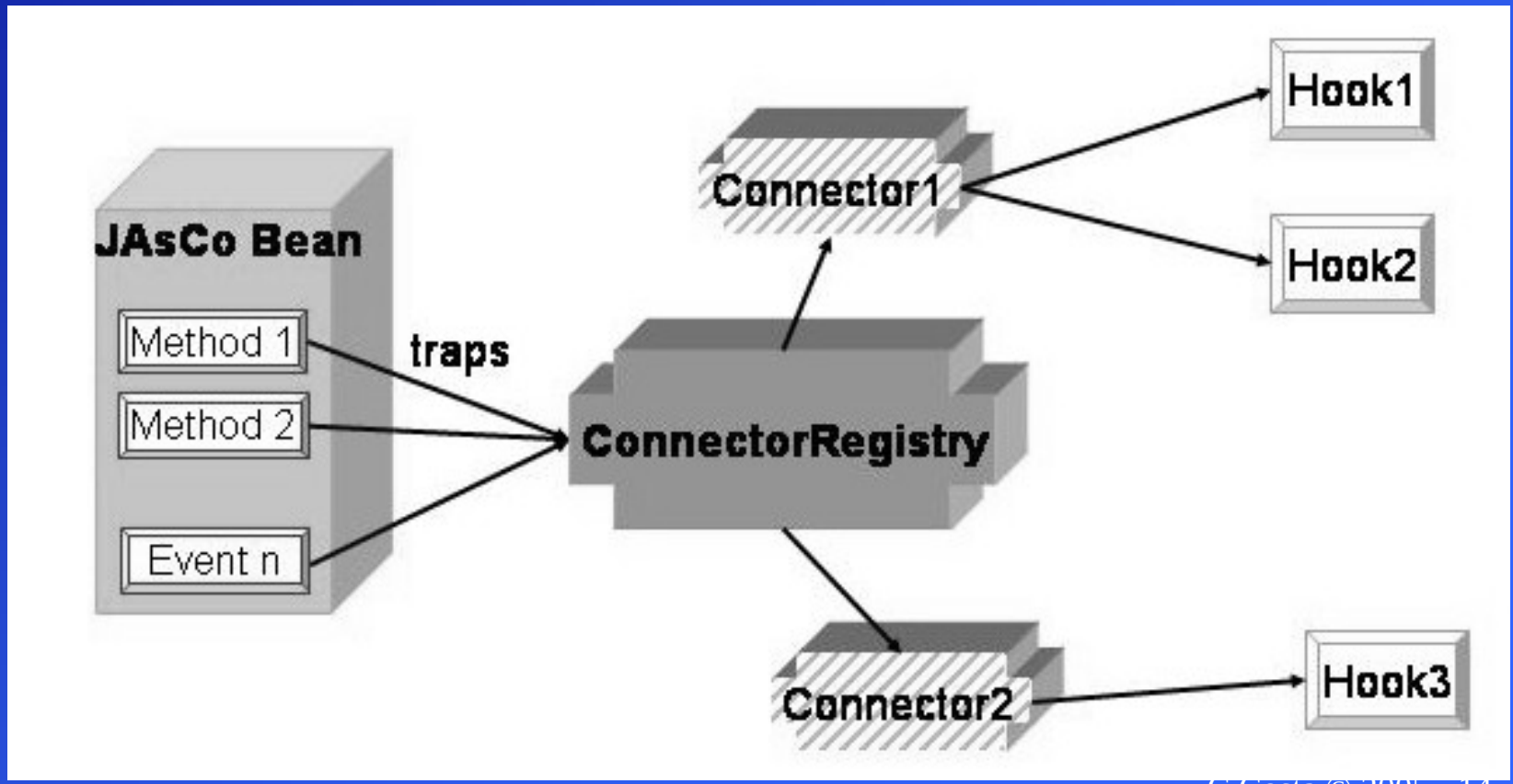
JAsCo.NET

- **Funcionalidad aportada (III):**



JAsCo.NET

- **Modelo de Componentes**



JAsCo.NET

- **Ventajas:**
  - Incorporación de aspectos en tiempo de ejecución
  - No necesita el código fuente de los componentes
  - Reutilización de aspectos y conectores
  - Capacidad expresiva
- **Inconvenientes:**
  - El registro de conectores no se realiza por código
  - La versión para .NET es aún prematura. Por ejemplo, el compilador de aspectos y conectores falla con los comentarios y con nombres de clases y métodos que utilicen el símbolo '_'
  - No se indica qué funcionalidad de la versión de Java (la más completa) ha sido implementada.

JAsCo.NET - Ejemplo (I)

Ejemplo de Aspecto

```
namespace Aspectos {

    class AspectoTest {

        hook ReemplazarHook {

            ReemplazarHook(string metodo()) {
                execute(metodo);
            }

            Replace() {
                MessageBox.Show("Metodo reemplazado!","AspectoTest");
                Console.WriteLine("Metodo reemplazado por AspectoTest.ReemplazarHook");
                return null;
            }
        }

        hook InformacionHook {

            InformacionHook (string metodo()) {
                execute(metodo);
            }

            Replace() {
                string original = (string)calledmethod.Invoke(calledobject,null);
                original = original + " - Interceptado por un aspecto";
                return original;
            }
        }
    }
}
```

JAsCo.NET - Ejemplo (I)

```
namespace Aspectos {
    using System.IO;
    using System.Collections;
    using Aspectos;

    namespace Conectores {

        static connector Reemplazar {

            AspectoTest.ReemplazarHook hook = AspectoTest.ReemplazarHook(
                 sring ProyectoJASCO.frmBase.EmitirMensaje() );

            hook.Replace();
        }
    }
```

Ejemplo de Conector1

de Aspecto

lazarHook");

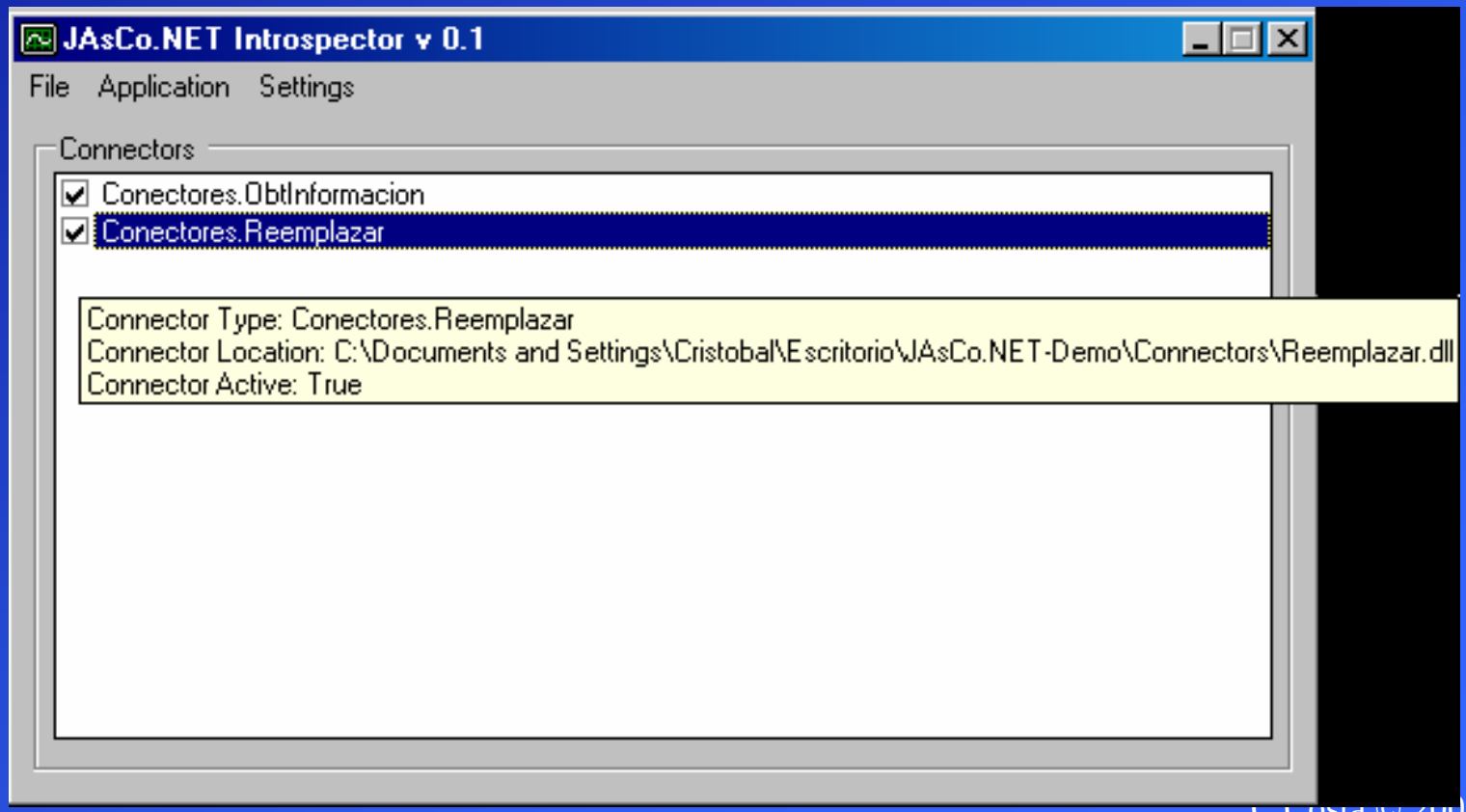
```
        hook I
            In
            }
            Re
            }
        }
    }
}
```

```
using System.IO;
using System.Collections;
using Aspectos;

namespace Conectores {

    static connector ObtInformacion {

        AspectoTest.InformacionHook hook = AspectoTest.InformacionHook(
            string ProyectoJASCO.frmBase.EmitirMensaje()
        );

        hook.Replace();
    }
}
```

Ejemplo de Conector2

JAsCo.NET - Ejemplo (II)

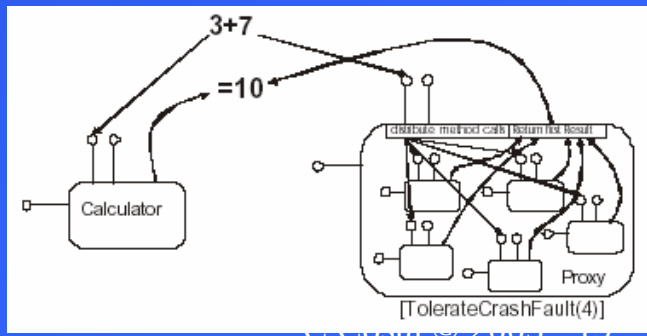
El *Introspector* es la aplicación que engancha/desengancha dinámicamente los conectores a la aplicación



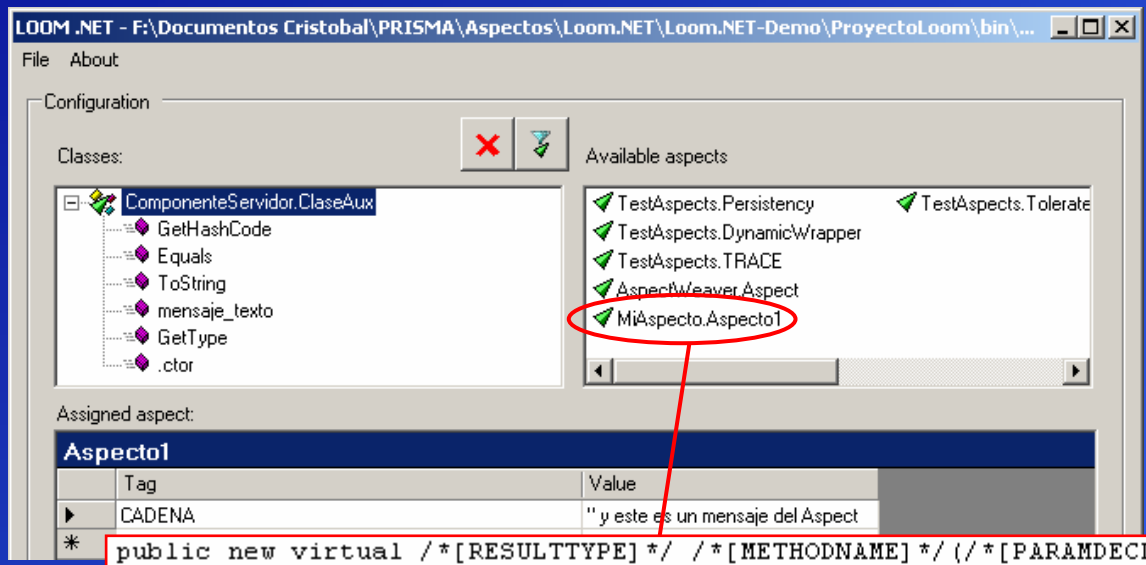
Loom.NET

http://www.dcl.hpi.uni-potsdam.de/cms/research/loom/

- Consta de dos proyectos:
  - Loom.NET → "Compilador" estático de aspectos
  - Rapier-Loom.NET → Compilador dinámico
- **Loom.NET**:
  - Los aspectos son reutilizables: se definen mediante *plantillas*
  - Los aspectos se unen al código mediante *proxys*. Se encapsula el código base y se crean unos componentes intermedios que deben ser llamados por la parte cliente.
  - Sólo puede enlazar un aspecto por clase
  - No permite conexión dinámica de aspectos



Loom.NET - Ejemplo



Weaver estático

Ejemplo de Plantilla

```
public new virtual /*[RESULTTYPE]*/ /*[METHODNAME]*/(/*[PARAMDECLARATION]*/)
{
    /*[RETVALINIT]*/
    System.Windows.Forms.MessageBox.Show("Ejecucion del aspecto en el contexto de /*[METHODNAME]*/" )
    /*[RETVALASSIGN]*/base./*[METHODNAME]*/(/*[PARAMLIST]*/) + " " + /*[CADENA]*/;
    System.Windows.Forms.MessageBox.Show("El mensaje ya se ha enviado hacia el cliente" );
    /*[RETVALRETURN]*/
}
```

Definición de los aspectos en XML

```
  <Aspect Name="MiAspecto.Aspecto1" Description="Simplemente engancha una cadena a otro metodo">
    <Tag Type="file" Name="CTOR">ctor_MiAspecto-Cadena.tpl</Tag>
    <Tag Type="file" Name="METHOD">method_MiAspecto-Cadena.tpl</Tag>
    <Tag Type="text" Name="BASECLASS">/*[BASENAMESPACE]*/./*[CLASSNAME]*/</Tag>
    <Tag Type="parameter" Name="CADENA">" y este es un mensaje del Aspecto ;-)"</Tag>
  </Aspect>
</Templates>
```

Rapier-Loom.NET

- No extiende el lenguaje. Los aspectos derivan de la clase *Aspect*, y la configuración se realiza mediante **atributos**.
- Instanciación **dinámica** de aspectos a través del operador *Weave.CreateInstance* que crea el aspecto unido a la clase destino.
- Características:
  - Permite asociar aspectos a Clases o Métodos
  - Enfatiza el uso de interfaces (permite extender la interfaz de la clase destino con la proporcionada por el aspecto, o enlazarse con métodos que implementen una determinada interfaz)
  - Operadores típicos: **Before, After, Instead, After Returning** (después de retornar de un método), **After Throwing** (después de lanzarse una excepción)

Rapier-Loom.NET

- Pueden engancharse varios aspectos a una misma clase
- Acceso al contexto de la clase objetivo
- Permite la comunicación entre aspectos, gracias al acceso al Contexto

- Inconvenientes:
  - No es posible desenganchar aspectos dinámicamente (pero se puede finalizar la instancia del objeto que lo tenía asociado)
  - No define la coreografía de llamadas para varios aspectos (aunque siguen el orden de creación)
  - Sólo se puede enlazar con métodos virtuales o definidos por una interfaz

Rapier-Loom.NET - Ejemplo

Código del Aspecto

```
// Hereda de Loom.Aspect
public class AspectoFuncional : Loom.Aspect {
    const string version = "AspectoFuncional v.01";

    // Estos atributos indican el nombre del método con el que debe enlazarse
    //    (la clase se especificará en el Weaving)
    [Loom.ConnectionPoint.Include("IniciarFuncionalidad")]
    [Loom.Call(Invoke.Instead)]
    // El código del aspecto propiamente dicho
    public string Identificarme() {
        string cadena;

        MessageBox.Show("Estoy en el metodo AspectoFuncional.Identificarme" +
            " y voy a llamar al CodigoBase", "AspectoFuncional");

        // Podemos acceder al contexto del objeto al que estamos enlazados
        // Accedemos a la clase base para modificar un atributo
        ((ClaseBase) Context.Instance).nombre = "¡¡¡Me han cambiado el nombre!!!";
        // Llamamos otra vez al código base para observar el resultado
        cadena=(Context.Invoke()).ToString() ;

        MessageBox.Show("Tras llamar a ClaseBase me ha devuelto: " + cadena,
            "AspectoFuncional");

        return "Y ahora me presento. Soy: " + version ;
    }

    // Ejemplo de función accesible desde fuera (sin weaving)
    //    De esta forma, el aspecto puede proporcionar funcionalidad adicional que pueda
    //    ser requerida por otras clases.
    public string Informacion() {
```

Rapier-Loom.NET - Ejemplo

Aspecto Distribución

```
// Hereda d
public clas
    const s

    // Esto
    //    (l
    [Loom.C
    [Loom.C
    // El c
    public
        str

        Mes


        //
        //
        ((C
        //
        cad

        Mes


        ret
    }

    // Ejem
    //    De
    //    se
    public
```

```
[Loom.ConnectionPoint.Include("IniciarFuncionalidad")]
[Loom.Call(Invoke.Instead)]
public string ObtenerDatosRemotos() {
    string cadena;
    Aspect[] listaAspectos;

    MessageBox.Show("Estoy en el metodo AspectoDistribucion.ObtenerDatosRemotos" +
        " y voy a pasarle un objeto al CodigoBase", "AspectoDistribucion");

    // Supongamos que se realiza una conexión con un componente remoto
    //    que nos devuelve una colección de datos. En este caso, cadenas
    ArrayList ObjetoCadenas = new ArrayList();
    ObjetoCadenas.Add("Hola");
    ObjetoCadenas.Add("Mundo");
    ObjetoCadenas.Add("!");

    // Ahora pasamos este objeto a la clase Base para su utilización
    ((ClaseBase) Context.Instance).objetoCompartido = ObjetoCadenas;

    // ¿Y si se lo pasamos directamente al aspecto funcional?
    MessageBox.Show("Le he pasado el objeto al CodigoBase, y ahora voy a pasarselo directam
        + " al Aspecto Funcional", "AspectoDistribucion");

    listaAspectos = Weaver.GetAspects(Context.Instance,
        typeof(Aspectos.AspectoFuncional));
    ((Aspectos.AspectoFuncional) listaAspectos[0]).Informacion2(ObjetoCadenas);

    // Llamada al codigo que se sustituye
    // Sino se llama, no se ejecutan los demás aspectos
    cadena=(Context.Invoke()).ToString();
    MessageBox.Show("ClaseBase me ha devuelto: " + cadena, "AspectoDistribución");
    return "Soy" + version + " y ya he realizado mis tareas";
}
```

Rapier-Loom.NET - Ejemplo

Unión de CódigoBase + Aspectos

```
[Loom.ConnectionPoint.Include("IniciarFuncionalidad")]
[Loom.Call(Invoke.Instead)]
// Aquí realizamos el Weaving con los Aspectos
public string LlamarComponenteConAspectos() {
    Aspect[] listaAspectos;
    string cadena;

    // Primero creamos los dos aspectos
    Aspectos.AspectoDistribucion dis = new AspectoDistribucion();
    Aspectos.AspectoFuncional asp = new AspectoFuncional();
    // Los metemos en un vector
    listaAspectos = new Aspect[2];  listaAspectos[0]=dis; listaAspectos[1]=asp;

    // Ahora enlazamos los aspectos con la clase Base
    // El orden de llamadas sigue el orden del vector, de forma que sólo se
    //   inicia el segundo si el primero le pasa el control al método que sustituye.
    ComponenteEstructural.ClaseBase obj =
        (ComponenteEstructural.ClaseBase) Loom.Weaver.CreateInstance(
            typeof(ComponenteEstructural.ClaseBase),null,listaAspectos);

    // Podemos acceder sin problemas a los aspectos enlazados con el objeto Base,
    //   y llamar a las funciones que se implementen.

        // Obtenemos los aspectos enlazados a un objeto
        listaAspectos = Weaver.GetAspects(obj,typeof(Aspectos.AspectoFuncional));
        // Ejecutamos código del aspecto enlazado
        cadena =((Aspectos.AspectoFuncional) listaAspectos[0]).Informacion();
        System.Windows.Forms.MessageBox.Show(cadena);

    // Ejecutamos el método de la clase Base que tiene código enlazado
    return obj.IniciarFuncionalidad();
}
```

SetPoint!

http://www.dc.uba.ar/people/proyinv/setpoint/

- Basado en la idea de la *Web Semántica*, introduce el concepto de **Descriptor Semántico** (***SetPoints***):
  - Marca que permite asignar una característica particular al código (o componente)
  - Los *JoinPoints* pasan a ser predicados lógicos sobre dichos descriptores
  - **Ejemplo**: asignamos la etiqueta [PROCESO_CRITICO] a un conjunto de métodos, podremos hacer:

    " Asociar el [aspecto_AltaPrioridad] a todos los métodos etiquetados como [Proceso_Critico] "

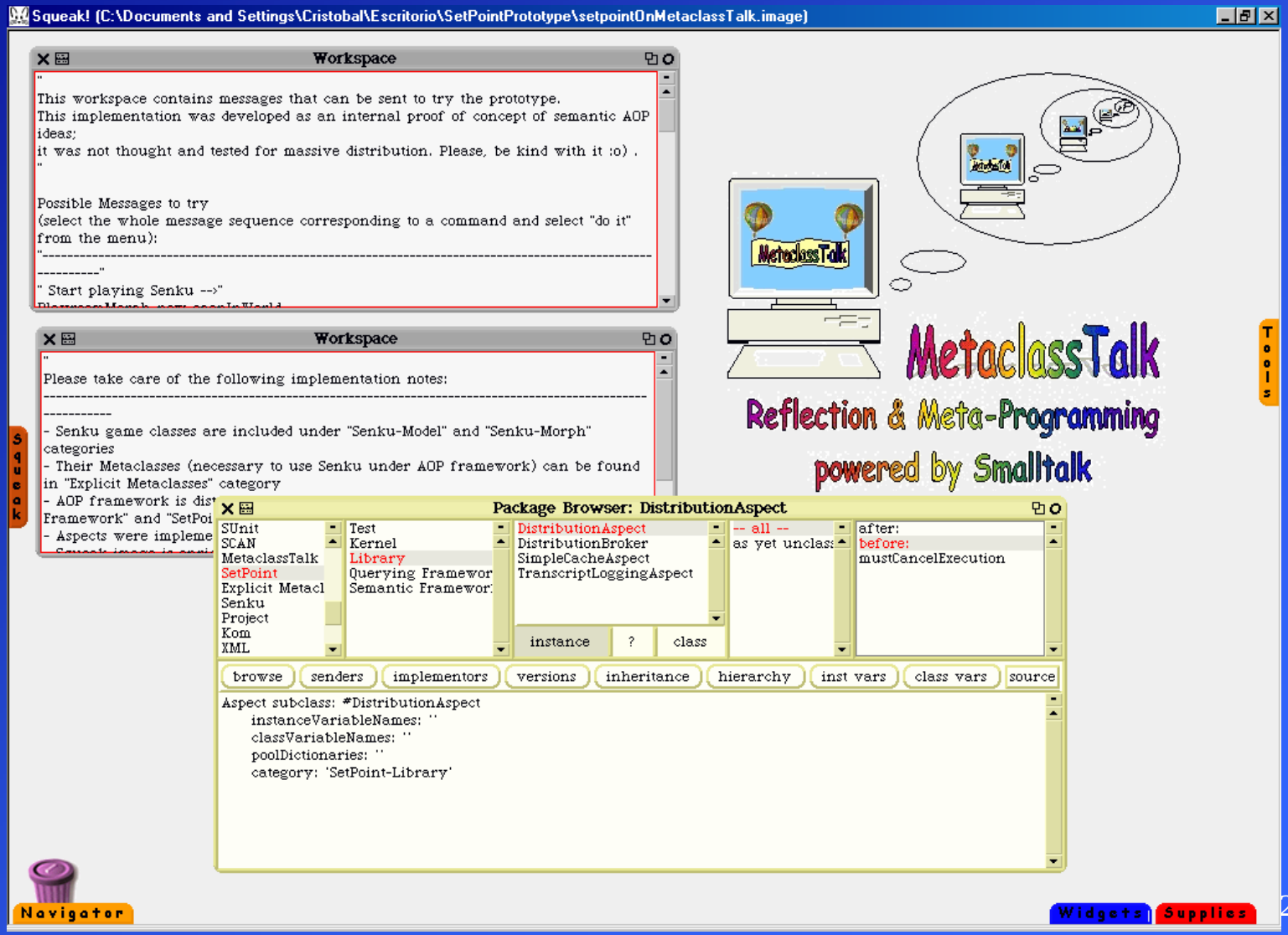
- Se consigue mayor independencia para establecer los *JoinPoints*
  - En las otras aproximaciones, los *JoinPoints* se declaran en base al nombre del método a enlazar (dependencia sintáctica):
  - **Ejemplo**: [Loom.ConnectionPoint.Include("Critico*")]

SetPoint!

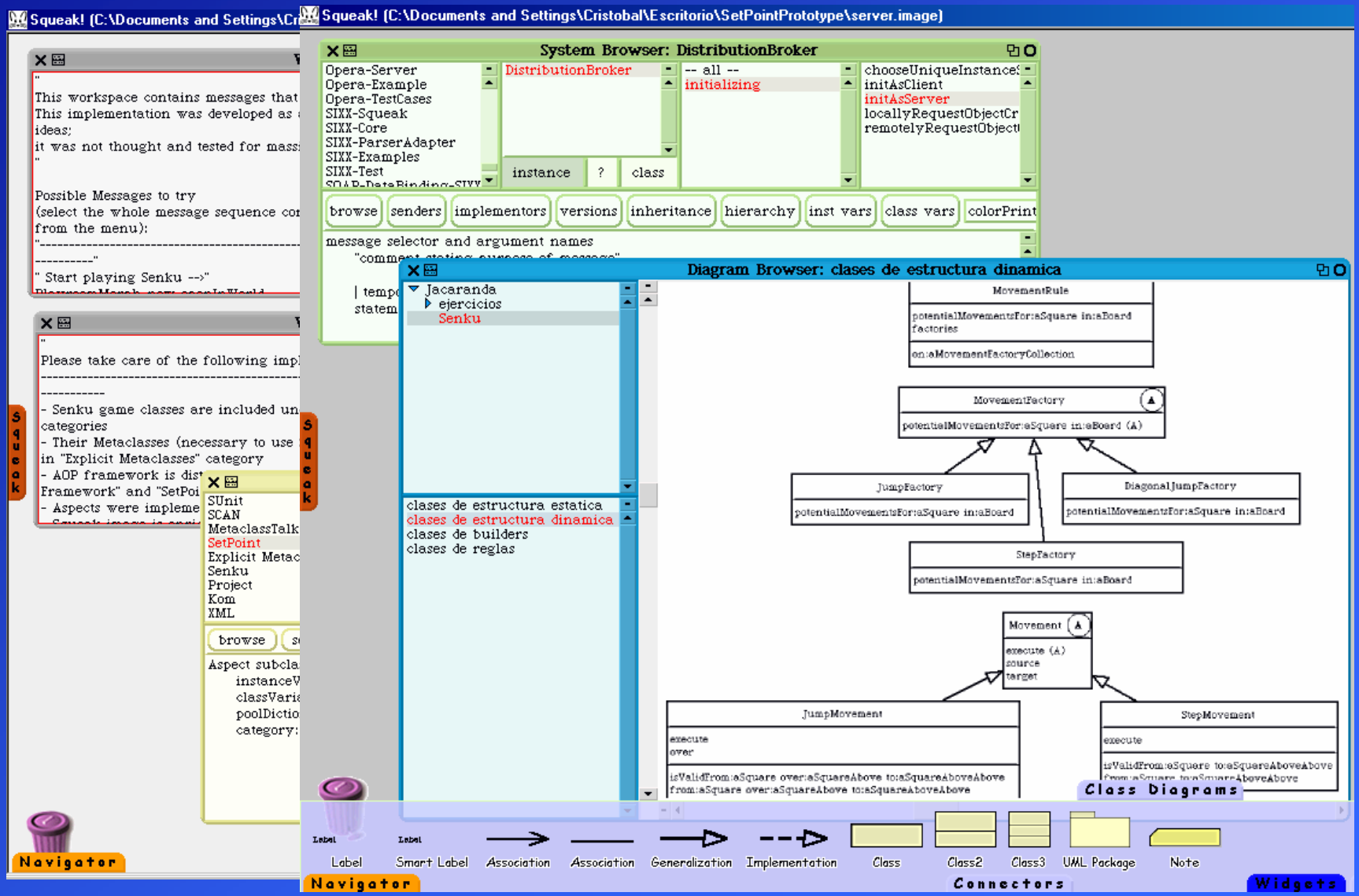
- Para dotarlos de mayor contenido semántico, se deberán utilizar ontologías para modelarlos adecuadamente (bien en la fase de diseño o en la fase de codificación)

- Proyecto en fase muy temprana. Se encuentran en la etapa de definir la sintaxis. No existe documentación.
- Existe un prototipo desarrollado en *SmallTalk*, que se ejecuta sobre **Squeak!,** una máquina virtual.

SetPoint! – Squeak!



SetPoint! – Squeak!



EOS

http://www.cs.virginia.edu/~eos/

- Aporta la posibilidad de enlazar aspectos a nivel de instancias (*instance-level aspects*)
  - Para ello, implementa el patrón *Mediator* mediante el uso de eventos
  - Weaving dinámico
- Los *Pointcuts* son definidos en los aspectos
- Permite la herencia de aspectos y asociar roles
- Acceso al contexto de ejecución, mediante mecanismos de reflexión:

  thisJoinPoint.{ getThis | getTarget | getReturnValue | getArgs }
- Requiere el código fuente de aspectos y código base (no soporta COTS)

Comparativa

	Alta reutilización	Weaving dinámico	Unión inclusiva de aspectos	Innovaciones
AspectC#	☑	☒	☑	
SourceWeave.NET	☑	☒	☑	Separación clara de código base y aspectos
Weave.NET	☑	☒	☑	Weavings definidos en un fichero XML
AspectDNG	☑	☒	☑	
Loom.NET	☑	☒	☒	Definición de aspectos con plantillas y expr. regulares
Rapier-Loom.NET	☒	☑	☑	Unión de aspectos al instanciarse el cód. base
SetPoint!	☒	☑	☒	Weavings definidos con predicados lógicos
EOS	☒	☑	☑	Aspectos a nivel de instancias
JAsCo.NET	☑	☑	☒	Lenguaje muy expresivo Relaciones entre aspectos