Tesis de Licenciatura
"Subjetividad en un Ambiente de Objetos"

Integrantes

Callegari, Claudia Estela (ccalleg@dc.uba.ar) L.U. 337/88
Eliaschev, Laura Sandra (eliaslau@hotmail.com) L.U. 98/89
Tanner, Carla Alejandra (ctanner@escape.com.ar) L.U. 1310/88

Director de Tesis

Lic. Máximo Prieto

Colaborador de Tesis

Lic. Pablo Victory

UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES

DEPARTAMENTO DE COMPUTACION

A nuestras familias,
Por el apoyo incondicional
Que nos brindaron durante toda la carrera.

Tesis de Licenciatura
"Subjetividad en un Ambiente de Objetos"

(APENDICES)

Integrantes

Callegari, Claudia Estela (ccalleg@dc.uba.ar) L.U. 337/88
Eliaschev, Laura Sandra (eliaslau@hotmail.com) L.U. 98/89
Tanner, Carla Alejandra (ctanner@escape.com.ar) L.U. 1310/88

Director de Tesis

Lic. Máximo Prieto

Colaborador de Tesis

Lic. Pablo Victory

UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES

DEPARTAMENTO DE COMPUTACION

Resumen / Abstract

Castellano

La Orientación a Objetos propone una manera de pensar que facilita la posibilidad de representar fielmente el mundo real en un ambiente computacional. Pero la realidad es mucho más compleja de lo que las herramientas de desarrollo orientadas a objetos actuales permiten modelar.

En el mundo real los “entes” están influenciados por factores que afectan su comportamiento, haciendo que ante diferentes situaciones respondan de manera distinta a un mismo requerimiento. Este problema es lo que justamente intenta resolver la “Subjetividad”.

El **objetivo** del trabajo es implementar “Subjetividad” en un ambiente de objetos, basándonos en el position paper "Real World Object Behaviour" [PRI/95].
Para tal fin elegimos “VisualWorks® Non-Commercial Release 3.0”. (Copyright © 1998 ObjectShare, Inc.)

Para cumplir este objetivo se define el siguiente **alcance**:

a. Definir, especificar y desarrollar las formas en que diferentes factores influyen el comportamiento de un objeto, y la manera en que estos factores interactúan.

b. Diseñar e implementar mecanismos para resolver de qué manera responder a un mensaje, de acuerdo con el factor que está influyendo el comportamiento de un objeto.

c. Determinar cómo influye la “Subjetividad” sobre los conceptos básicos de la Orientación a Objetos, analizando también su impacto dentro del ambiente a desarrollar.

d. Modificar el ambiente de trabajo seleccionado para permitir el manejo de “Subjetividad”. Esto implica:
   - Desarrollar el motor de “Subjetividad”.
   - Modificar las herramientas utilizadas para construir aplicaciones para que los programadores habituales de Smalltalk puedan manejar el ambiente subjetivo de una manera natural.

English

Object-Orientation proposes a way of thinking that facilitates the modeling of the real world. But the real world is much more complex than the current development tools allow us to model.

In the real world objects are influenced by different factors which affect how they behave; in this way, the objects answer differently to the same message on different occasions. This problem is what “Subjectivity” tries to solve.

The **goal** of this work is to implement “Subjectivity” in an object environment, basing our ideas on the position paper “Real World Object Behaviour”. [PRI/95]
We selected “VisualWorks® Non-Commercial Release 3.0” (Copyright © 1998 ObjectShare, Inc.) object environment to do this development.

In order to fulfill this goal we define the following **scope** for the project:

a. Specify and develop the ways in which the different factors influence an object behaviour, and the way in which these factors interact.

b. Design and implement mechanisms to solve how to respond to a message according to the factor that influences the object.

c. Specify how “Subjectivity” affects the Object-Oriented basic concepts, analizing the impact in the environment under development.

d. Modify the programming environment selected to deal with “Subjectivity”. This implies:

- To develop a subjective engine.
- To modify tools in order to allow the programmers to deal with the subjective environment in a friendly an natural way.

Introducción

Antes de comenzar queremos mencionar que hemos realizado diferentes Apéndices donde dejamos sentados los conceptos básicos sobre los cuales nos basamos para realizar la tesis; entre ellos se incluyen los conceptos básicos de la programación orientada a objetos, el lenguaje UML utilizado para modelar el diseño y el lenguaje de programación Smalltalk.

¿Qué es la Subjetividad?

Motivación de la Subjetividad

Dos de los grandes beneficios de la orientación a objetos es que permite modelar fielmente el mundo real y al mismo tiempo es una alternativa de programación más flexible. Pero el comportamiento complejo de los objetos reales es tal que no es posible modelarlos en forma natural con las herramientas disponibles en la actualidad. Esto genera una distancia entre el mundo real y los modelos computacionales. A partir de esta distancia es que se originan muchos problemas de diseño. Si bien los patterns de diseño encaran estos problemas e intentan resolverlos, no los eliminan de raíz. Nos preguntamos: ¿Por qué surgen estos problemas de diseño? Uno podría pensar que es porque se equivocó al diseñar o porque el modelaje de objetos nos llevó a eso. Sin embargo, creemos que estos problemas surgen por una limitación de los ambientes y herramientas de desarrollo actuales. La Orientación a Objetos es mucho más rica que ellos. [PRI/97]

Para que los modelos computacionales se acerquen al mundo real debemos disminuir la distancia existente entre ambos, permitiendo a nuestros modelos incluir otra dimensión en ellos: la “Subjetividad” de las cosas.

Como comienzo de este trabajo creemos que lo más conveniente es poner en claro qué se entiende por el término de “Subjetividad”.

Si revisamos la bibliografía sobre el tema, rápidamente llegaremos a la conclusión que no existe aún un sentido cohesivo de lo que es exactamente la Subjetividad [FOO/95]. Muchos investigadores parecen proyectar sus propios intereses y necesidades sobre la definición de Subjetividad. Al mismo tiempo, cada definición parece variar de acuerdo con el método de diseño y el lenguaje que se utiliza para resolver el problema. [HAR/95].
Consideramos que la definición de Subjetividad debe ser lo suficientemente general como para soportar cualquier dominio de problemas, y además ser independiente de la tecnología utilizada para implementarla.
Por este motivo, nuestra definición de Subjetividad surge de la observación del mundo real, donde los objetos se comportan de diferente manera de acuerdo con las diferentes situaciones en las que se ven involucrados. Veamos esto con algunos ejemplos.

Cuando una persona conduce un auto deportivo se comporta de manera distinta a cuando conduce un camión, más allá de que las operaciones básicas sean las mismas.

Análogamente las personas responden de modo diferente a una misma pregunta dependiendo de quién la esté realizando (nótese que uno no utiliza la misma modalidad –tono de voz, formalidad de la respuesta, lenguaje, etc.- si el interlocutor es su jefe, su esposo, o un policía).

En el ámbito bancario, una cuenta puede estar en diferentes estados: habilitada (para operatoria normal), inhabilitada (por falta de fondos, por orden judicial, etc.), o cerrada. Dependiendo de estos estados, una misma operación realizada sobre la cuenta tendrá resultados diferentes. Por ejemplo: Una extracción de fondos de una cuenta habilitada llevada a cabo en un cajero automático, resultará en la entrega del dinero requerido, el descuento de dicho monto de la cuenta, y la devolución de la tarjeta. En cambio, una

extracción de fondos de una cuenta inhabilitada por orden judicial, puede resultar en la retención de la tarjeta sin haber entregado el efectivo.

En una interfase gráfica de usuario (GUI), las ventanas pueden encontrarse abiertas, cerradas, representadas como íconos, minimizadas, etc. El envío del mensaje "refresh" a una ventana, podría provocar que la misma necesite chequear el modo en el que se encuentra, para determinar qué comportamiento adoptar.

En este punto cabe aclarar que los ejemplos explicados anteriormente deben ser considerados como una primera aproximación al tema de "Subjetividad", y por lo tanto hay que tomarlos como analogías de este tema ya que, por estar contados en un lenguaje natural, no están planteados formalmente como objetos que se envían mensajes, tal como ocurre en un ambiente de objetos.

Nuestra definición de "Subjetividad"

El hecho de que ante una misma situación los objetos se comporten distinto en el mundo real no es dependiente de un dominio de problema. Como se dijo anteriormente, ocurre en el ámbito bancario, en los sistemas de computación, en la comunicación entre las personas, etc. Esta variación de comportamiento está regida por diferentes factores que influyen a los objetos. En los ejemplos previos, estos factores están representados por la esposa o el jefe que hace la pregunta, el tipo de vehículo que se conduce, el estado de la cuenta bancaria y el modo en el que se encuentra la ventana.

Basándonos en esta diferencia de comportamiento (que denominaremos **Comportamiento Subjetivo**) es que definimos a la Subjetividad de la siguiente manera:

> La **Subjetividad** es *"la habilidad que tiene un objeto de comportarse de diferentes maneras al momento de recibir un mensaje, de acuerdo con los estímulos o factores (internos o externos) que influyen sobre él en dicho momento"*.

En un ambiente de objetos, el comportamiento de un objeto está definido por su protocolo, o sea: los mensajes que es capaz de recibir este objeto, y no por los mensajes que es capaz de emitir. Por este motivo nuestra definición no considera lo que le sucede al objeto emisor de un mensaje, y se orienta solamente a la variación de comportamiento del objeto receptor.

Analicemos las siguientes situaciones dentro del ámbito bancario:

- El titular de una cuenta corriente consulta el saldo de la misma, y en base a ello decide si realizar un depósito para incrementar dicho saldo.
- Un empleado del banco consulta el saldo de la cuenta corriente, para realizar un ranking de clientes y tomar posteriormente una decisión gerencial.
- Un proceso de impresión de resúmenes consulta el saldo de la cuenta corriente, para emitir el resumen que le será enviado posteriormente al titular de la cuenta a través del correo postal.

Como podemos observar en estas tres situaciones, la cuenta corriente recibe el mensaje "saldo", y siempre responde con el mismo comportamiento. Sin embargo, cada uno de los emisores de este mensaje (el titular, el empleado del banco y el proceso de impresión), realiza acciones diferentes con la respuesta obtenida, fruto de distintas necesidades. Este es el punto que deseamos resaltar: el hecho de que los tres emisores tengan un comportamiento diferente frente a la respuesta recibida, está dado porque sus necesidades son diferentes, y para nosotros esto NO es un comportamiento subjetivo, ya que en un ambiente de objetos el comportamiento se define en función de los mensajes que un objeto es capaz de recibir, y no de emitir. Por este motivo, para nosotros no existe subjetividad en estas situaciones.

Hasta aquí hemos definido lo que para nosotros es la Subjetividad. Nos tomamos el permiso de utilizar esta definición, ya que si bien no existe un consenso generalizado sobre lo que es la Subjetividad, la postura aquí descripta fue expuesta durante los workshops relacionados con este tema organizados en la "Object-Oriented Programming Systems, Languages and Applications" (OOPSLA) de los años 1995 y 1996, siendo aceptada como una aproximación válida. [PRI/95] [PRI/96] [HAR/95]

Historia de la Subjetividad

En primer término nos parece importante mencionar los diferentes trabajos e investigaciones que se realizaron con respecto al tema de la "Subjetividad". Para ello intentaremos realizar una cronología que nos permita ubicarnos en cómo fue desarrollándose y obteniendo mayor importancia el tema en cuestión.

Una breve cronología

Año 1990
- Barbara Pernici presenta su paper "*Object with Roles*" [PER/90] en la Conferencia de Office Information Systems, introduciendo el tema de roles.

Año 1992
- Ivar Jacobson edita su libro "*Object-Oriented Software Engineering*" [JAC/92] y hace referencia al tema de roles que pueden desempeñar diferentes actores cuando describe los casos de uso de un sistema, y cómo un actor (mismo objeto) puede tener varios roles.

Año 1993
- Junio: Antero Taivalsaari escribe "*Object-oriented programming with modes*" [TAI/93]
- Septiembre: Se realiza "Object-Oriented Programming Systems, Languages and Applications" (OOPSLA) en Washington, D.C. (USA). Harrison y Ossher presentan su paper "*Subject-Oriented Programming (A Critique of Pure Objects)*" [HAR/93], ***marcando el comienzo explícito del tema "Subjetividad"***.

Año 1994
- Octubre: Se realiza "Object-Oriented Programming Systems, Languages and Applications" (OOPSLA) en Portland, Oregon (USA). Durante esta conferencia se lleva a cabo el ***primer workshop sobre el tema de "Subjetividad"***.

Año 1995
- Octubre: Se realiza "Object-Oriented Programming Systems, Languages and Applications" (OOPSLA) en Austin, Texas (USA). Aquí Máximo Prieto y Pablo Victory presentan el position paper sobre el cual basamos esta tesis, denominado "*Real World Object Behavior*" [PRI/95].

Año 1996
- Hafedh Mili, W. Harrison y H. Ossher, escriben el paper *"Supporting Subject-Oriented Programming in Smalltalk*", donde describen un prototipo que H.Mili realizó en Smalltalk basándose en el paper escrito por Harrison y Ossher en 1993. [MIL/96]
- Máximo Prieto y Pablo Victory publican su position paper "*Subjectivity: Towards True Polymorphism*", continuando con su trabajo del año anterior. [PRI/96]

Año 1997
- Máximo Prieto y Pablo Victory publican su artículo "*Subjective Object Behavior*". [PRI/97]

Año 1998
- Febrero: John Vlissides publica en la revista "C++ Report" su artículo denominado "*Pattern Hatching. Subject-Oriented Design*".

Año 1999
- Noviembre: Se realiza "Object-Oriented Programming Systems, Languages and Applications" (OOPSLA) en Denver, Colorado (USA). H.Ossher y Peri Tarr presentan su paper "*Using Subject-Oriented Programming to Overcome Common Problems in Object-Oriented Software Development*

and Evolution".

Un resumen de la historia

Durante los años previos a 1993, si bien no existía una definición exacta de "Subjetividad", ya se había detectado la necesidad de la misma, y habían comenzado algunos trabajos que podrían considerarse antecesores a este tema.

En **1990** Barbara Pernici introduce la noción de "roles" en su paper "Object with Roles". [PER/90] En el mismo plantea un modelo para describir el comportamiento de objetos basado en el concepto de roles.
El planteo de roles surge debido a la complejidad de capturar aspectos dinámicos del comportamiento de un objeto. Un rol describe diferentes perspectivas para la evolución del mismo. Este concepto permite que un objeto reciba y envíe diferentes mensajes en diferentes niveles de evolución. Por ejemplo, una persona en una oficina puede desempeñar acciones bajo diferentes roles: empleado de una compañía, director de uno o más departamentos, miembro de una familia, estudiante de una universidad, etc. En cada uno de estos roles la persona ejecuta acciones pertinentes a ese rol.

Un objeto puede jugar diferentes roles en diferentes momentos y también puede jugar más de un rol en un mismo momento.
El modelo planteado por Pernici le permite al diseñador describir en forma separada el comportamiento de los objetos en cada rol y relacionar los diferentes comportamientos en diferentes roles especificando las reglas y restricciones que gobiernan el comportamiento concurrente.

Formalmente, desde este nuevo punto de vista, una clase pasa a modelarse de la siguiente manera:

clase = (cn,
{R_0 = <rn_0, P_0, S_0, M_0, RU_0 >,
{R_1 = <rn_1, P_1, S_1, M_1, RU_1 >,
.....
{R_v = <rn_v, P_v, S_v, M_v, RU_v >})

donde cn es el nombre de la clase, y los R_i's son los *tipos de roles* que describen diferentes comportamientos de un objeto de una clase durante su ciclo de vida. Una clase contiene descripciones de diferentes roles, cada uno de los cuales puede ser jugados por los objetos de una clase. Cada objeto tiene un *tipo de rol base*, R_0, el cual describe las características iniciales de un objeto en su creación y las propiedades globales concernientes a su evolución. Cada tipo de rol R_i consiste de: el nombre de un rol rn_i, un conjunto de propiedades P_i, un conjunto de estados S_i, un conjunto de mensajes M_i y un conjunto de reglas RU_i. Definamos a continuación cada uno de los componentes del rol.

Propiedades: Una propiedad p_i □ P_i es una descripción abstracta de un dato asociado con un objeto y su implementación por una variable de instancia.

Estados: Cada estado s_i □ S_i define uno de los posibles estados en los que se puede encontrar el rol y determina el contexto actual de ejecución de un objeto en un rol determinado.

Mensajes: Cada mensaje $m_i \in M_i$ es un mensaje de entrada o de salida que el objeto puede recibir o enviar a través de los cambios de estados en el rol considerado.

Reglas: Una regla ru_i □ RU_i define qué mensajes pueden ser enviados y recibidos en cada rol y cuáles son las transiciones de estado permitidas. Para cada mensaje enviado o recibido se asocia una regla de transición de estados. O sea: las reglas describen el comportamiento de un objeto dentro del rol, expresando las posibles secuencias de estados.

Como vemos este modelo es una primera solución al problema de capturar el comportamiento complejo y dinámico que presentan los objetos. Si bien este trabajo es previo a la aparición formal del tema de "Subjetividad", más adelante veremos que algunos autores se basan en este concepto para desarrollar la Subjetividad en ambientes orientados a objetos.

En **1992** Ivar Jacobson [JAC/92] ya visualiza los roles desde la etapa de análisis como un hecho natural que sucede en el mundo real y plantea la utilidad de la detección de actores y roles para definir los casos de uso.

En el mes de **Junio de 1993**, Antero Taivalsaari escribe su artículo denominado "Object-oriented programming with modes" [TAI/93]. En el mismo plantea el concepto de "modos" (o "estados lógicos") en los que se pueden encontrar los objetos, y deja plasmada teóricamente una posible extensión del modelo convencional orientado a objetos basado en clases, de tal forma que soporte la definición explícita de estos "modos".

Taivalsaari expresa que durante el diseño y la implementación de los programas, uno a menudo se encuentra con objetos que pueden alcanzar diferentes estados o condiciones, no refiriéndonos a los valores de las variables de dichos objetos, sino a una clase de estado más conceptual. Por ejemplo: en una interface gráfica de usuario (GUI), las ventanas pueden estar abiertas, cerradas, representadas como íconos, etc. A estos estados lógicos son a los que Taivalsaari denomina "modos". Veamos más detenidamente su planteo.

Él pudo observar que la presencia de "modos" en los programas es muy frecuente y rara vez se le presta atención, aunque es obvio que tales estados lógicos son importantes en el diseño y la implementación de los programas, ya que los métodos suelen variar considerablemente dependiendo del "modo" en el que se encuentre el objeto en dicho instante. Y además de ser esenciales en la implementación de abstracciones, los "modos" son claramente esenciales en el diseño, especificación y análisis de los componentes del programa. Sin embargo, aunque la programación orientada a objetos enfatiza la importancia de la abstracción –la habilidad de separar la esencia de los detalles – la potencialidad de los "modos" como una facilidad de abstracción había sido ignorada hasta ese momento, ya que casi todos los lenguajes orientados a objetos hasta ese entonces forzaban a que la información de los estados lógicos de los objetos estuvieran ocultos en las operaciones de dichos objetos (por ejemplo incluyendo sentencias del estilo "if-case" para testear algún estado interno del objeto y decidir en base a ello qué rama de operaciones ejecutar). Esto provocaba que las operaciones incluyeran construcciones condicionales con el solo propósito de asegurar que se elija la porción adecuada de comportamiento en una situación en particular. Esto no era muy satisfactorio, ya que el uso excesivo de sentencias condicionales hacía que las definiciones de las operaciones tendieran a ser más largas y complicadas de leer y mantener, haciendo decrecer la claridad conceptual de los programas y forzando a que la información esencial a nivel de diseño se ocultara en la implementación de las operaciones.

Para afrontar este tema Taivalsaari propone utilizar la noción de "modos" (como una construcción explícita del lenguaje), y permitir que una clase posea múltiples implementaciones de una misma operación.
Para cada "modo" se definen un conjunto de operaciones en forma separada, de tal manera que en tiempo de ejecución la selección de la operación esté basada en el conjunto de operaciones correspondientes al "modo" en que se encuentre el objeto. Junto con esta idea se plantea un esquema que permite manejar el cambio de estados de un objeto, utilizando lo que denomina una "función de transición", la cual maneja automáticamente el cambio de estados. Estas "funciones de transición" son operaciones definidas exclusivamente para determinar el modo en el que se encuentra el objeto (testeando por ejemplo valores de las variables de instancia, de otros objetos del sistema, etc.). Estas "funciones de transición" no pueden ser invocadas explícitamente sino que su ejecución es manejada automáticamente por el sistema.

A continuación transcribiremos los dos ejemplos que Taivalsaari explica en su artículo.

Primero presentamos el ejemplo de la ventana, donde se supone que las ventanas de un sistema pueden aparecer en la pantalla de tres modos diferentes: abiertas (open), cerradas (closed) o presentadas como íconos (iconified). Como puede observarse, en este ejemplo aún no introduce la noción de "funciones de transición".

```
CLASS Window IMPLEMENTATION IS

        (* define properties which are common to all modes *)
        VAR locX, locY: Integer;          (* location *)
        VAR sizeX, sizeY: Integer;        (* size *)
        VAR contents: Printable;

        MODE open IS                      (* define properties of opened windows *)
                METHOD open               IS (* no operation – already open *) END;
                METHOD refresh            IS ...update the window on the screen... END;
                METHOD close              IS ...make window disappear...; become closed; END;
                METHOD iconify            IS self.close; become iconified; self.refresh; END;
        ENDMODE open;

        MODE closed IS                    (* define properties of closed windows *)
                METHOD open               IS become open; self.refresh; END;
                METHOD refresh            IS (* no operation -- window is closed *) END;
                METHOD close              IS (* no operation -- already closed *) END;
                METHOD iconify            IS become iconified; self.refresh; END;
        ENDMODE closed;

        MODE iconified IS                 (* define properties of iconified windows *)
                METHOD open               IS self.open; become open; self.refresh; END;
                METHOD refresh            IS ...draw the icon on the screen... END;
                METHOD close              IS ...make icon disappear...; become closed; END;
                METHOD iconify            IS (* no operation -- already iconified *)  END;
        ENDMODE closed;

        INITIALIZATION IS                 (* initialize a new instance of Window *)
                sizeX:= 200; sizeY:= 150;(* initialize size *)
                locX:=50; locY:=50;       (* initialize location *)
                become closed;            (* initialize mode *)
        ENDINIT;

ENDCLASS Window;
```

Figura 1: Implementación parcial de la clase “Window” presentada por Taivalsaari, cuando introduce la noción de modos [TAI/93]

Veamos ahora el ejemplo del Stack donde Taivalsaari explica también las funciones de transición.

```
CLASS Stack [T] INTERFACE IS

        MODES empty, non-empty, full;
        SIGNATURE
                push:    T->0;
                pop:     0->T;
                top:     0->T;
        ENDSIGN;
ENDCLASS;


CLASS Stack [T] IMPLEMENTATION IS

        VAR Items: Array;
        VAR Sp: Integer;

        MODE empty IS
                METHOD push (item:T)  IS Sp:= Sp + 1; Items[Sp]:= item; END;
                METHOD pop:T          IS RAISE_ERROR "Stack empty"; END;
                METHOD top: T         IS RAISE_ERROR "Stack empty"; END;
        ENDMODE empty;

        MODE non-empty IS
                METHOD push (item:T)  IS AS_IN_MODE empty END;
                METHOD pop:T          IS result := Items[Sp]; Sp := Sp -1; END;
                METHOD top: T         IS result := Items[Sp]; END;
        ENDMODE non-empty;

        MODE full IS
                METHOD push (item:T)  IS RAISE_ERROR "Stack full"; END;
                METHOD pop:T          IS AS_IN_MODE non-empty; END;
                METHOD top: T         IS AS_IN_MODE non-empty; END;
        ENDMODE non-empty;

        TRANSITION IS (* the transition function *)
                empty           WHEN Sp = 0;
                full            WHEN Sp = Items.size;
                non-empty       OTHERWISE;
        ENDTRANS;

        INITIALIZATION (size: Integer) IS
                Items   := Array.new(size);
                Sp      := 0;
        ENDINIT;

ENDCLASS;
```

Figura 2: Implementación de un Stack, que ilustra las funciones de transición propuestas por Tailvaalsari [TAI/93]

El enfoque de trabajo con "modos" requiere que el programador piense sobre el comportamiento de una abstracción en cada situación posible. En este sentido, este enfoque mejora la seguridad del código y disminuye la distancia ("gap") entre el diseño y la implementación de los programas.

Como puede observarse, si bien Taivalsaari no lo menciona explícitamente en su trabajo, está claramente refiriéndose a la "Subjetividad" de las cosas, planteando una realidad que no se puede obviar, e intentando ofrecer una solución teórica al problema (o parte del mismo).

En **Septiembre de 1993** se realiza "Object-Oriented Programming Systems, Languages and Applications" (OOPSLA) en Washington, DC. Durante la misma, William Harrison y Harold Ossher presentan su paper denominado "Subject-Oriented Programming (A Critique of Pure Objects)" [HAR/93], trabajo que marca el comienzo explícito del tema "Subjetividad". En este paper los autores proponen la denominada *"Programación Orientada a Sujetos",* cuyo objetivo es facilitar el desarrollo y la evolución de conjuntos de aplicaciones cooperantes. En este contexto son importantes los siguientes requerimientos:

- Debe ser posible desarrollar aplicaciones separadas y luego componerlas, siendo estas aplicaciones independientes una de otras no importando el grado de cooperación.
- Debe ser posible la introducción de una nueva aplicación a una composición sin requerir modificaciones a las otras aplicaciones integrantes de la composición, y sin invalidar los objetos persistentes ya creados por ellas.
- Deben soportase todas las aplicaciones no previstas, inclusive las extensiones de las ya existentes.
- Dentro de cada aplicación deben conservarse las ventajas de encapsulamiento, herencia y polimorfismo.

En el paradigma de orientación a sujetos descripto por Harrison y Ossher, cada aplicación es un sujeto o una composición de sujetos, un término que define una colección de especificaciones de estado y comportamiento asociados a una aplicación en sí. Estos sujetos reflejan una pequeña y delimitada percepción del mundo real, porque "son programas (o fragmentos de programas) orientados a objetos que modelan su dominio a su propio modo: de manera subjetiva", ya que modelan las diferentes "vistas subjetivas" que tienen sobre otros objetos.

Harrison y Ossher usaron el ejemplo de un árbol en el bosque, el cual puede ser manipulado desde el punto de vista de un pájaro construyendo su nido, o desde el punto de vista de un leñador, o también desde el punto de vista de un asesor de impuestos. En el modelo clásico orientado a objetos, el pájaro, el leñador y el asesor de impuestos son clientes del árbol, y acceden a él a través de mensajes.

Analizando un poco el modelo clásico, en el momento de diseñar el árbol, se trabaja con las propiedades esenciales del árbol y los comportamientos que afectan a estas propiedades. Por ejemplo, las propiedades esenciales del árbol podrían ser: la altura, el peso, la cantidad de hojas, la masa, etc. Y su comportamiento: crecer, fotosintetizar, etc. Al aparecer en este mundo otro objeto, como el asesor de impuestos, éste elabora su propia vista de las características y comportamientos asociados al árbol. En este caso, las características de esta vista del árbol son la contribución del árbol al valor asegurado del lugar en el que crece. El comportamiento incluye los métodos para calcular esta contribución. Estos métodos pueden variar de un tipo de árbol a otro. Es más, pueden formar parte de la vista del asesor de impuestos de todos los objetos, árboles y no-árboles. Como hemos notado, estas características y comportamientos son externos al árbol en sí mismo, formando parte de la *vista subjetiva que el asesor de impuestos tiene del árbol*. Análogamente sucede lo mismo con el leñador y el pájaro, como puede observarse en la siguiente figura:

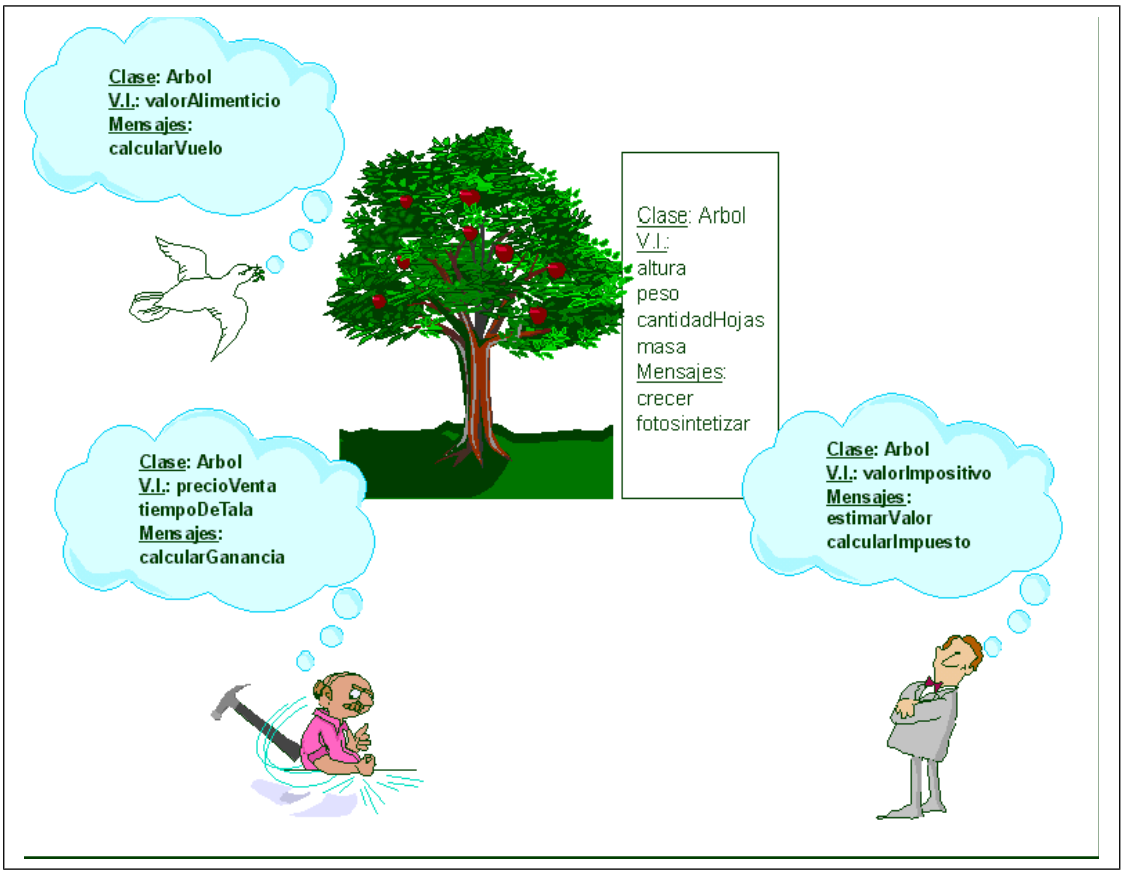


Figura 3: Diferentes Vistas Subjetivas de un Arbol orientado a objetos (según Harrison y Ossher)

En términos generales, el problema que tratan Harrison y Ossher se refiere a que un mismo objeto puede ser manipulado por distintas aplicaciones y cada una de éstas requiere que el objeto soporte operaciones particulares. La solución propuesta por los autores es modelar las aplicaciones como diferentes vistas subjetivas de los objetos.
El modelo clásico de objetos es, en muchos aspectos, el modelo de objetos visto por cualquier sujeto. La Programación Orientada a Sujetos incluye a la Programación Orientada a Objetos como uno de sus elementos tecnológicos.

Un ***sujeto*** es una colección de especificaciones de estado y comportamiento que reflejan una estructura particular, una percepción del mundo, tal como las que son vistas por una aplicación o herramienta particular. Estos sujetos reflejan una pequeña y delimitada percepción del mundo real, porque "son programas (o fragmentos de programas) orientados a objetos que modelan un dominio a su propio modo: de manera subjetiva", ya que modelan las diferentes "vistas subjetivas" que tiene sobre otros objetos. Este modelo subjetivo está definido por la jerarquía de clases del sujeto, que contiene solamente los detalles implementados y/o utilizados por el sujeto.

Bajo este modelo, un sujeto es un paquete que contiene la información de varias clases de objetos que trabajan juntos para alcanzar un objetivo particular. Cada sujeto provee sus propias definiciones de clases, reflejando el punto de vista sobre la cual se basa el sujeto.

Una ***activación de sujeto*** –a menudo conocida solamente como ***activación***-, provee una instancia de ejecución para el mismo, incluyendo la manipulación actual de los datos por el sujeto en cuestión.

Los autores utilizan el término ***identificador de sujeto*** para hacer referencia a la identificación única de un objeto que aparece en el contexto de uno o más sujetos de interés.

Los sujetos pueden componerse y producir nuevos sujetos, y el comportamiento funcional del nuevo sujeto está controlado por la especificación de una serie de ***reglas*** que describen las correspondencias y la forma de composición de los elementos que componen el sujeto. Al momento de componer sujetos deben examinarse cuidadosamente los sujetos y sus interrelaciones, para determinar cómo serán combinados. Esto incluye: correspondencia de operaciones, de clases, de variables de instancias y de métodos. Existen justamente dos tipos básicos de reglas:

- ***regla de correspondencia***: especifica el significado de los nombres en el sujeto compuesto.

- ***regla de composición***: establece cómo se combinan los correspondientes elementos de los sujetos que forman parte de la composición del nuevo sujeto.

La interacción entre los sujetos puede realizarse de diferentes formas:

- Un pedido de funcionalidad o cambio de estado a ser provisto por otro sujeto. Por ejemplo, la invocación de "talar" por parte del leñador afecta la altura del árbol y también un consumo calórico por parte del leñador.

- Realización de una actividad en la cual otro sujeto puede participar. Por ejemplo, la estimación del impuesto realizada por el asesor puede ser reflejada por una interpretación privada de la decisión del leñador para estimar el esfuerzo de talar el árbol.

- Notificación de una ocurrencia que puede ser de interés para otro sujeto. Por ejemplo, la actividad de construcción del nido por parte del pájaro podría estar influenciada por el plan que tenga el leñador de talar el árbol.

- Uso del comportamiento de un sujeto como parte de un comportamiento más complejo de otro sujeto. Por ejemplo, el asesor de impuestos puede utilizar al leñador para talar varios árboles para cobrar una deuda.

- Compartir información de estado. Por ejemplo, el pájaro y el leñador podrían tener la misma noción de la altura del árbol.

Todo el código en un framework orientado a sujetos ejecuta en el contexto de una activación en particular. De esta forma, una invocación a una operación puede ser modelada como una tupla:

(a, op, p)

donde:

- *a* es la activación del sujeto que realiza la invocación.
- *op* identifica a la operación que debe ser realizada.
- *p* es la lista de parámetros.

Cuando en un modelo orientado a sujetos se invoca una operación, ésta puede causar la ejecución de métodos en múltiples sujetos. Las reglas de composición se encargan de controlar esto.

Bajo el modelo propuesto por Harrison y Ossher, la Programación Orientada a Sujetos permite acomodar la característica del mundo real en la cual diferentes sujetos clasifican a los objetos en diferentes jerarquías. Por ejemplo, el pájaro podría clasificar los objetos en plantas (proveedoras de nectar, proveedoras de insectos, etc.), anidables, y predadores. Mientras tanto, el leñador podría clasificar a los

objetos en árboles (de madera dura, de madera blanda) y no árboles. Estas clasificaciones representan dos formas diferentes de mirar las cosas, como podemos ver gráficamente en el siguiente diagrama.

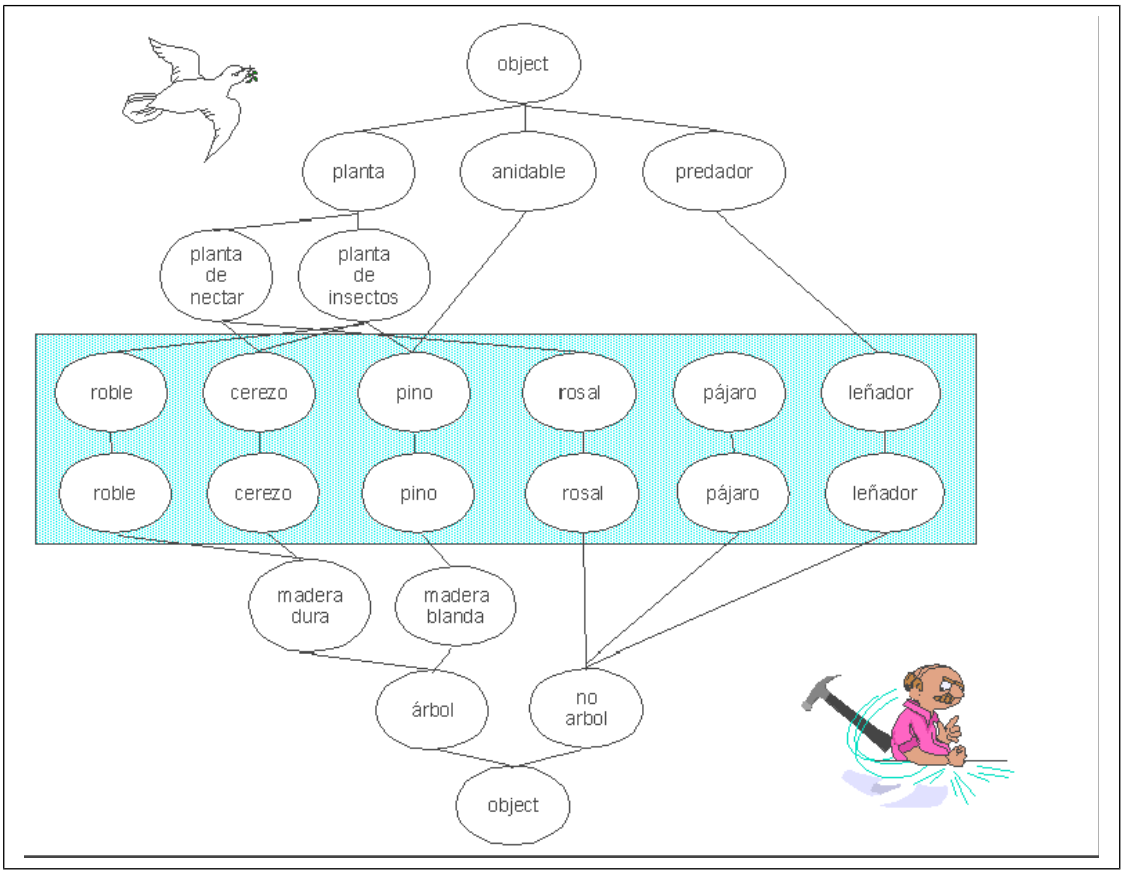


Figura 4: Dos jerarquías de clases utilizadas por diferentes sujetos: leñador y pájaro (según Harrison y Ossher)

La Programación Orientada a Sujetos propuesta por Harrison y Ossher intenta proveer una solución al manejo de estas situaciones, instanciando sujetos y haciendo que ruteen mensajes en forma adecuada.

Luego de este paper, los autores continuaron realizando trabajos relacionados con este tema; entre otras cosas, definieron con más detalle las reglas de correspondencia y composición, hicieron definiciones más formales sobre ellas, y básicamente ampliaron los temas abordados en el paper que se acaba de comentar.

En **Octubre de 1994** se lleva a cabo “Object-Oriented Programming Systems, Languages and Applications” (OOPSLA) en Portland, Oregon (USA). Aquí se organiza el primer workshop relacionado al área de “Subjetividad”.
Por tratarse del primer workshop, su objetivo fue facilitar la discusión exploratoria de la Subjetividad e identificar algunos de los conceptos y conclusiones claves.

David Ungar y Randall Smith resumieron el soporte para la subjetividad brindado por el lenguaje Us que le permite a un objeto comportarse en forma diferente dependiendo de la "perspectiva" utilizada por el cliente. [HAR/94]

Hayden Lindsey hizo una revisión de los problemas del desarrollo de sistemas que motivan el interés en el tema de subjetividad. [HAR/94]

Dave Cleal presentó la necesidad de soportar la subjetividad para mejorar la reusabilidad. [HAR/94]

Michael Vanhilst describió su experiencia en la construcción de un visualizador astronómico. El diseño original había sido procedural y sufría de varios problemas al momento de realizar mejoras. Intentó reconstruirlo utilizando la tecnología de Objetos sin resultados prometedores. Las clases se tornaron muy grandes y la funcionalidad útil tuvo que ser removida porque el modelo Orientado a Objetos la hacía muy difícil de implementar. Un ejemplo es los mapeos muchos-a-muchos de ventanas de imágenes. Durante esa época, él estaba intentando utilizar el enfoque subjetivo para resolver estos problemas. Encontró un número de características útiles para la composición de sujetos: componiendo agregaciones, aumentando los atributos, cambiando el comportamiento de métodos mediante el agregado de implementaciones, y componiendo clases con mapeos parciales. [HAR/94]

Harrison y Ossher discutieron el prototipo que estaban construyendo para soportar subjetividad y composición de sujetos, conocido con el nombre de WASP (El WAtson Subject-composition Prototype), el cual se basa en la postura presentada por ellos en el año 1993 y que se explicara previamente en este trabajo. [HAR/94]

Craig Chambers describió las formas en las que Cecil contiene soporte de subjetividad. Cecil es un lenguaje puro orientado a objetos, con multi-métodos. Los multi-métodos son típicamente declarados bajo diferentes alcances, pero fuera de las clases en sí mismas. Otros aspectos también pueden ser declarados en forma separada, por ejemplo atributos. Los mecanismos de multi-métodos y alcances son introducidos para localizar extensiones a bibliotecas individuales para propósitos de encapsulamiento y simplicidad. [HAR/94]

A lo largo de las diferentes presentaciones realizadas durante la “Object-Oriented Programming Systems, Languages and Applications” (OOPSLA) se plantearon diversos conceptos generales con respecto al tema de Subjetividad para que pudieran ser considerados en futuras discusiones e investigaciones.

En el mes de **Octubre de 1995**, se llevó a cabo “Object-Oriented Programming Systems, Languages and Applications” (OOPSLA) en Austin, Texas (USA).
Aquí se desarrolló el segundo workshop de Subjetividad, que fue estructurado para cubrir una serie de tópicos, sobre los cuales los participantes pudieron explorar sus dudas, resultados, especulaciones, conclusiones, etc. Los tópicos de las áreas fueron: Requerimientos de Aplicaciones, Principios, Soporte de Subjetividad en el Mundo Real, Conexiones entre vistas subjetivas de un objeto, e Interacción de Usuarios con Objetos Subjetivos.

Bent Kristensen explicó su intención de utilizar roles directamente como parte de un estilo de modelización y luego discutir el mapeo de ideas a diferentes lenguajes de programación.

Kristensen define un rol de un objeto como “un conjunto de propiedades, las cuales son importantes para el objeto, para permitirle comportarse de cierta manera esperada por un conjunto de otros objetos”. [HAR/95]

La idea de roles es modelar una abstracción que permita manejar una determinada perspectiva de un objeto. Los roles pueden aparecer o desaparecer durante el tiempo de vida de un objeto. Se los puede asociar a una propiedad de un objeto, agregando operaciones o extendiendo el comportamiento del objeto. De igual forma, se los puede asociar a propiedades de otros roles.

Pablo Victory expuso el position paper “Real World Object Behavior” [PRI/95], sobre el cual basamos este trabajo. Hubo un acuerdo generalizado en cuanto a que un despacho del estilo multi-método era una buena cosa para hacer, pero también hubo inquietudes acerca de la comprensibidad del sistema

resultante.

Michael Werner explicó la razón por la cual él considera conveniente la utilización de Subjetividad en las bases de datos. En su visión, diferentes sujetos cooperan unos con otros con el objetivo de crear una base de datos compartida. La cooperación implica que los objetos serán creados o modificados por un sujeto y accedidos (leídos) por otro. Cada sujeto individual puede percibir el esquema de base de datos a su propio modo: una vista de un sistema de base de datos es utilizada por varios actores, cada actor jugando diferentes roles, y cada rol puede contener varios casos de uso. Cada actor, rol y caso de uso ve una porción (vista) de la base de datos total. [HAR/95]

En el año **1996**, Hafedh Mili, Harrison y Ossher presentaron su paper "Supporting Subject-Oriented Programming in Smalltalk", donde describen un prototipo que Mili realizó en Smalltalk para soportar subjetividad, basándose en el enfoque presentado por Harrison y Ossher en 1993 e implementado posteriormente en C++. En este trabajo se transfirieron muchas de las soluciones desarrolladas para C++ directamente a Smalltalk. [MIL/96]

Durante el mismo año, Máximo Prieto y Pablo Victory publican su position paper "Subjectivity: Towards True Polymorphism", continuando con su trabajo del año anterior [PRI/96] y presentando una manera para extender el concepto de polimorfismo a nivel de objeto. Para conseguir esta extensión, el objeto debe ser subjetivo al ambiente que lo rodea. Cuando recibe un mensaje debe ser capaz de elegir una de varias implementaciones diferentes para contestar al mismo.

Los mismos autores publican en el año **1997** su artículo "Subjective Object Behavior" [PRI/97] donde amplían los conceptos de fuerzas y de desarrollo en un ambiente subjetivo, marcando las ventajas de la utilización de tales ambientes para una organización.
Parte de nuestra tesis consistió en desarrollar los conceptos expuestos en estos últimos dos trabajos. Los mismos son tratados en detalle a lo largo de nuestra tesis.

En **Febrero de 1998**, John Vlissides (uno de los autores del libro "Design Patterns: Elements of Reusable Object Oriented Software" [GAM/95]) publica en la revista "C++ Report" un artículo denominado "Pattern Hatching: Subject-Oriented Design" [VLI/98], en el cual analiza la visión de Subjetividad presentada por Harrison y Ossher [HAR/93]. Según John Vlissides, un buen diseño orientado a objetos es aquel que se puede modificar y extender agregando código en lugar de invadirlo, haciendo el cambio aditivo y no invasivo. De esta forma el cambio resulta más fácil, más localizado, tiene menos errores y es más mantenible que uno invasivo. La mayoría de los patterns de diseño puede ayudar a que los cambios invasivos sean innecesarios, o al menos, poco necesarios. Pero aún así, algunos patterns no lo logran.

Vlissides remarca que utilizando Subjetividad se puede extender el código, ayudando a eliminar los cambios invasivos, mejorando el uso de patterns. El autor concluye que las facilidades de la Programación Orientada a Sujetos inician una nueva dimensión en el diseño de software. Hay potencialmente muchos problemas que se pueden solucionar con la Subjetividad. Es importante considerarla como un mecanismo y no como una solución en sí, debiendo entender cómo aplicarlo a los problemas desde el punto de vista del diseño de software. Es un nuevo mecanismo que probablemente generará idiomas y patterns propios. Por ahora, hay que aprender cómo los sujetos nos pueden ayudar a implementar, mejorar y quizás obviar algunos patterns de diseño de hoy.

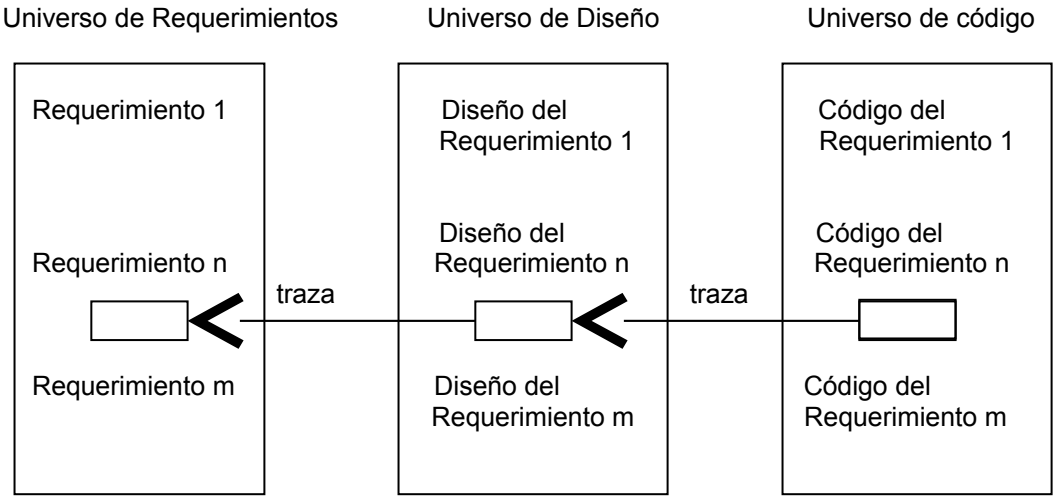
En la "Object-Oriented Programming Systems, Languages and Applications" (OOPSLA) realizada durante el año **1999** en Denver, Colorado (USA), Harold Ossher y Peri Tarr retoman el trabajo de John Vlissides para orientar el trabajo que ellos venían realizando sobre Subjetividad [HAR/93] hacia metodologías de diseño. Este trabajo lo presentan en su paper "Using Subject-Oriented Programming to Overcome Common Problems in Object-Oriented Software Development/Evolution" [OSS/99]. Aquí presentan un

enfoque diferente del diseño de sistemas, basado en la composición y descomposición flexible provista por la Programación Orientada a Sujetos, alineando de esta forma el diseño con las especificaciones de requerimientos y con el código. Los autores muestran cómo este enfoque permite que los beneficios del diseño se mantengan a través del ciclo de vida del sistema.

Para lograr ésto, los modelos de diseño estándares se descomponen en unidades potencialmente solapables y más pequeñas denominadas "sujetos de diseño". Cada uno de estos sujetos de diseño encapsula una pieza de funcionalidad coherente y simple diseñada desde su propia perspectiva y modelada usando construcciones de diseño orientadas a objeto estándares. Los sujetos de diseño pueden ser compuestos de diferentes maneras produciendo fragmentos de diseños completos o de gran escala. De manera similar, la Programación Orientada a Sujetos permite que el código orientado a objetos sea descompuesto en sujetos. Cada sujeto de diseño es refinado en un sujeto de código y los detalles de la composición del código pueden ser derivados a partir de los detalles de la composición del diseño. A este nivel, la trazabilidad resulta excelente porque los sujetos de diseño y de código se corresponden directamente. (Se entiende por "***trazabilidad***" a la habilidad de determinar cómo se corresponden los requerimientos, diseños, códigos, planes de testeo, etc., y cómo se propagan de una manera consistente los cambios a lo largo de las distintas piezas relacionadas).

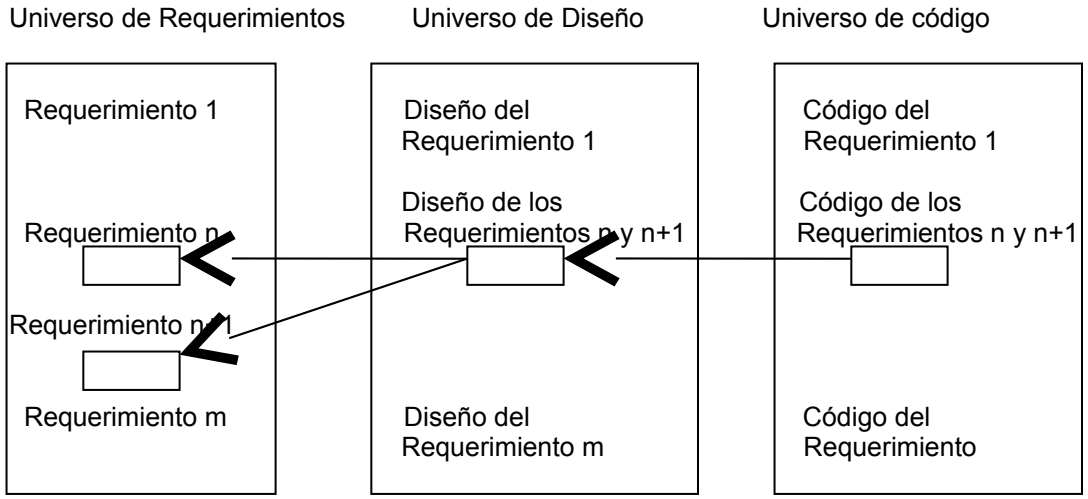
Veamos gráficamente las correspondencias entre las especificaciones de requerimientos, elementos de diseño y elementos de código, y cómo resulta la trazabilidad en cada caso.

- La alineación entre las especificaciones de requerimientos, los elementos de diseño y los elementos de código es directa: un elemento de código se corresponde con un elemento de diseño y este a su vez, se corresponde con una especificación de requerimientos.



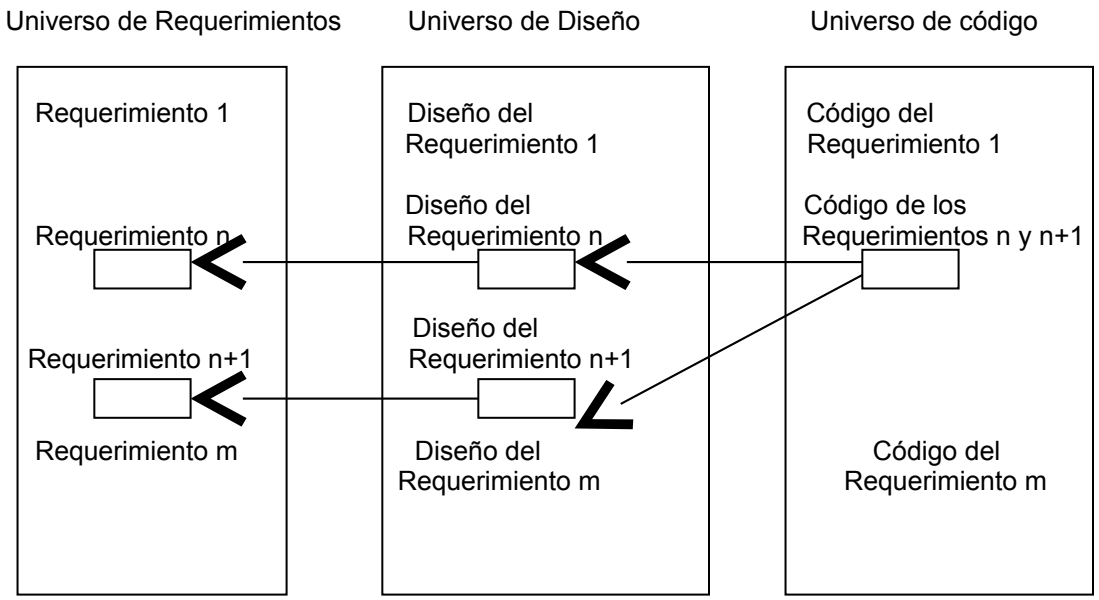
Una modificación en el "Requerimiento n" sólo tiene impacto en los elementos correspondientes al mismo.

- La alineación entre las especificaciones de requerimientos, los elementos de diseño y los elementos de código no es directa: un elemento de diseño se corresponde con más de una especificación de requerimientos.



Una modificación en el "Requerimiento n" puede tener impacto en el "Requerimiento n+1", ya que alcanza a elementos de diseño en común de ambos requerimientos.

- La alineación entre las especificaciones de requerimientos, los elementos de diseño y los elementos de código no es directa: un elemento de código se corresponde con más de un elemento de diseño.



Una modificación en el "Requerimiento n" puede tener impacto en el "Requerimiento n+1", ya que alcanza a elementos de código en común de ambos requerimientos.

A partir de esta historia, puede notarse que el tema de Subjetividad es bastante nuevo. Esto hace que no exista todavía un sentido cohesivo de lo que realmente es la Subjetividad, o más específicamente, cómo se desea y necesita que sea. Por eso muchos participantes de los Workshops nombrados anteriormente parecían proyectar sus propios intereses y requerimientos en el tema.

Tal planteo resulta poco deseable y nos atreveríamos a decir que peligroso, ya que nos estamos enfrentando a un área que recién está emergiendo. Esta situación nos indica que estamos en el comienzo del proceso a través del cual puedan surgir los intereses y premisas comunes sobre la Subjetividad.

Dado que nosotros no estamos limitadas a plantear una solución para un dominio de aplicación específico, nuestra intención es dar una especificación e implementación de Subjetividad lo suficientemente general para soportar cualquier dominio de problema, dejando sentadas las bases para luego definir las metodologías de diseño.

Enfoque de la tesis

Esta tesis propone un enfoque diferente al tema de la Subjetividad. Se basa en el concepto que los objetos varían su comportamiento de acuerdo con factores o estímulos que los influyen, tal como sucede en la vida real. De aquí en más denominaremos a estos factores con el nombre de “fuerzas”.

La idea es que un objeto reaccione de distintas formas a la recepción de un mismo mensaje, de acuerdo con la fuerza que lo esté afectando en ese momento. Para esto debemos permitir que un objeto tenga distintas implementaciones (métodos) para el mismo mensaje. Cuando un objeto recibe un mensaje debe determinar qué implementación utilizar para responder el mensaje recibido, teniendo en cuenta la fuerza predominante al momento de recepción del mensaje. Entonces, bajo la fuerza predominante, selecciona la implementación correspondiente.

El mecanismo de decisión utilizado para decidir bajo qué circunstancias predomina una fuerza, y cuál de las implementaciones utilizar, debería estar especificado de alguna manera por el programador. Este mecanismo hace que el objeto reaccione de manera subjetiva a la recepción de un mensaje.

Desarrollo

Primero plantearemos un caso de estudio que nos permita establecer un marco de referencia en este trabajo. Luego comenzaremos con el desarrollo de nuestra visión de la Subjetividad, dividiéndola en dos áreas: el planteo teórico y la implementación del ambiente subjetivo. Por último expondremos nuestras conclusiones.

Antes que nada, vale la pena aclarar que nuestro trabajo se encuentra dirigido a un ambiente basado en clases (no a prototipos), y es por ello que todo el desarrollo y las consideraciones realizadas a lo largo de la tesis asumen esta postura.

Un Caso de Estudio: El Stack

A menudo, en muchos libros y papers, encontramos que el “Stack” (o “Pila”) es tomado como base para diferentes ejemplos. Esto se debe a que es conceptualmente muy fácil de entender, y debido al uso popular de los mismos dentro del área de sistemas, casi todos lo conocen, permitiendo además muchas extensiones y analogías.

Por este motivo no es casualidad que tomemos al “Stack” como nuestro caso de estudio. Veamos entonces cómo se relacionan un “Stack” y la “Subjetividad”.

Un **Stack** es un objeto que almacena una colección de elementos. Permite agregar y eliminar elementos con la única restricción que se trate del tope del stack. Esto es la disciplina LIFO (Last-In-First-Out). Típicamente un stack implementa los mensajes “push”, “pop”, “top” y “isEmpty”. El método “push” agrega un elemento al stack. El método “pop” retorna el tope del stack eliminándolo del mismo. El método “top” retorna el elemento que se encuentra en el tope del stack sin sacarlo. El método “isEmpty” testea si el stack está vacío. Este stack no tiene límite superior en cuanto a la cantidad de elementos que puede almacenar.

Un **Stack Limitado** es un stack, con la diferencia que sí posee un límite superior en cuanto a la cantidad de elementos que es capaz de almacenar.

Veamos ahora una típica implementación de estas clases. Podemos pensar que Stack es subclase de Object, y LimitedStack es subclase de Stack. Si bien sabemos que esta decisión de diseño no es la más conveniente pues no es bueno subclasificar una clase de otra clase concreta, tomaremos la subclasificación mencionada de este modo por los siguientes motivos:

1) Por cuestiones didácticas y de simplicidad al momento de explicar los ejemplos.

2) Porque la mayoría de los libros suele presentar esta clasificación para explicar este ejemplo.

Observemos que en realidad, lo más conveniente sería contar con una clase abstracta de la cual se subclasifiquen el Stack y el LimitedStack. De esta manera, si en un futuro deseamos realizar modificaciones y/o extensiones al comportamiento definido en la clase Stack, esto no afectaría al comportamiento definido en la clase LimitedStack. (notar que esto no es así en la subclasificación propuesta actualmente).

Habiendo realizado esa aclaración, ahora podemos decir que la clase LimitedStack agrega el método “isFull” para testear si el stack se encuentra lleno, y sobreescribe el método “push” definido en Stack, para agregarle al mismo la restricción correspondiente a la cantidad máxima de elementos.

En la siguiente figura podemos ver gráficamente el diagrama de clases y las implementaciones de los mensajes más relevantes para los próximos ejemplos.

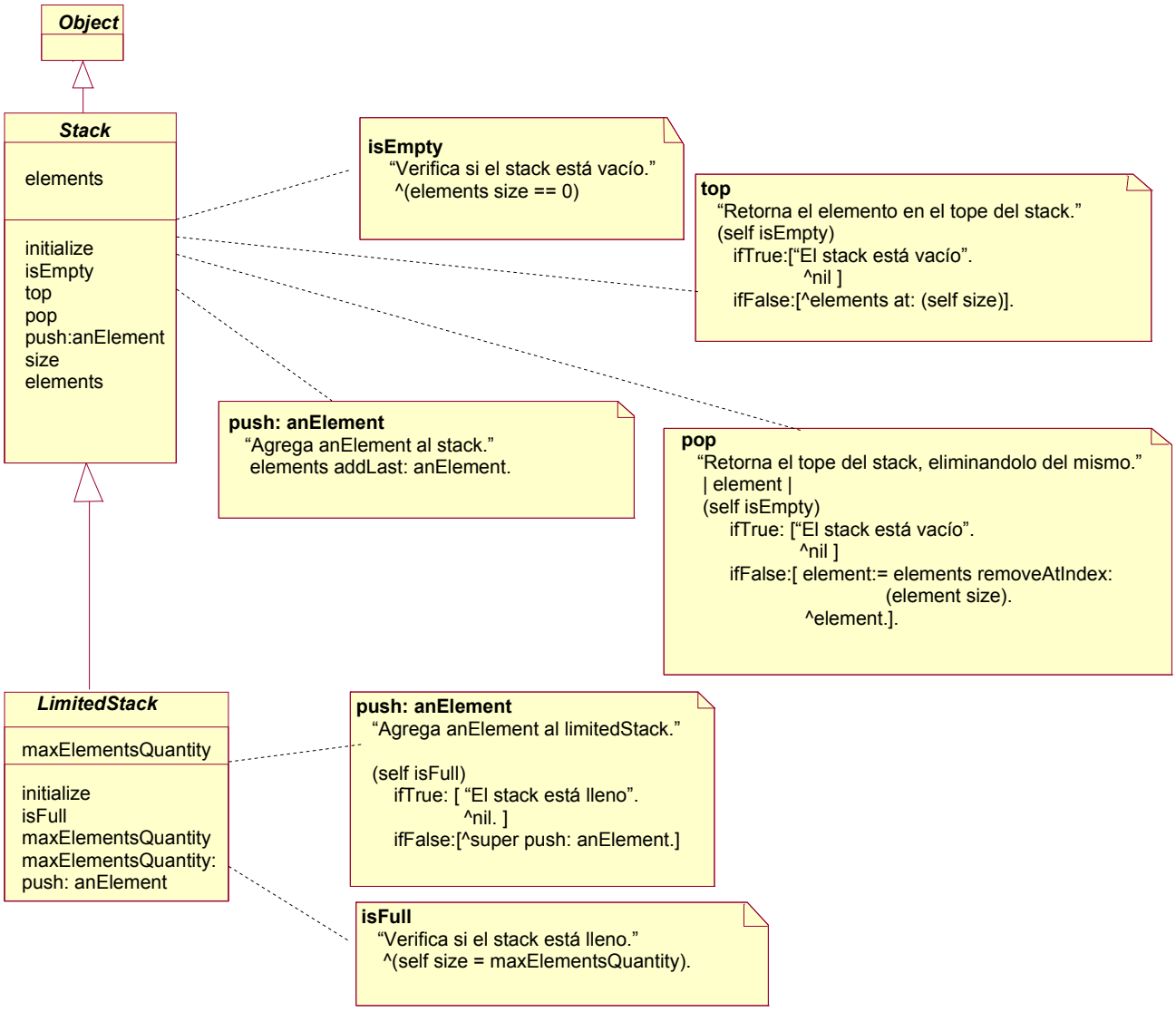


Figura 5 : Diagrama de Clases para Stack y LimitedStack

Hasta ahora, no hemos dicho nada con respecto a la Subjetividad del Stack. A modo de ejemplo, observemos detenidamente la implementación del método "pop" de la clase "Stack". Como podemos ver existe una sentencia condicional, ya que esta operación que retorna el tope del stack es válida siempre y cuando el stack no esté vacío. Si miramos esto de otro modo, podríamos decir que el stack varía su comportamiento de acuerdo con la cantidad de elementos existentes en el mismo. ¿ Cómo cambiaría esta situación si contáramos con "Subjetividad" ?. La idea sería contar con dos implementaciones

diferentes para el método “pop” (una para cuando el stack está vacío y otra para cuando no lo está), y que el stack, al recibir el mensaje “pop” elija de algún modo cuál es la implementación adecuada que debe ejecutarse de acuerdo con la cantidad de elementos del mismo. Para el emisor del mensaje este mecanismo sería totalmente transparente, pero el stack estará respondiendo de manera distinta a un *mismo* mensaje de acuerdo con las condiciones dadas. Esto puede verse como comportamiento subjetivo.

Lo mismo sucede para los métodos “top” de “Stack”, y “push” de “LimitedStack”.

Más adelante analizaremos en forma más profunda y detallada este ejemplo. La idea aquí es mostrar un caso sencillo donde puede verse la “Subjetividad”, y cuál es la idea que perseguimos.

Nuestra Visión de la Subjetividad

Cómo influye la Subjetividad a los conceptos básicos de la Orientación a Objetos

Si observamos detenidamente nuestro alrededor nos daremos cuenta de que los objetos de este mundo, (incluídos nosotros mismos), varían el comportamiento de acuerdo con las circunstancias en las cuales están inmersos. Desde un auto que arranca o no su motor dependiendo de la cantidad de combustible en su tanque de nafta hasta la respuesta a un saludo de acuerdo con la persona que lo emite, muchos objetos responden de diferente manera a un determinado mensaje más allá de que las operaciones básicas sean las mismas.

Pero también es cierto que no todos los objetos de este mundo varían su comportamiento ante diferentes situaciones. Por ejemplo: los números no poseen un comportamiento subjetivo; siempre responden de la misma manera a un mismo mensaje. Esto fortalece la idea de que no se debe forzar un ambiente puramente subjetivo sino que es conveniente poder contar con un ambiente mixto, o sea: un ambiente donde el programador decida si para una determinada clase de objetos hace uso o no de la “Subjetividad”, de tal forma que permita contar con una clara extensión del paradigma original y no con una restricción del mismo. Además es muy importante que la extensión a realizar no fuerce a tener que reconvertir código ya escrito.

Comenzando a explorar el tema de Subjetividad, comienzan a surgir las primeras preguntas...

¿ Qué significa que un objeto sea subjetivo? ¿ Y qué significa hablar de un mensaje subjetivo? ¿Y de una clase subjetiva?

Clases y Mensajes Subjetivos

Decimos que un **objeto** es **subjetivo** si es capaz de responder de distintas maneras a un mismo mensaje (de acuerdo a determinadas circunstancias). Esto es análogo a decir que dichos objetos poseen **comportamiento subjetivo**, es decir, que en su protocolo contienen uno o más mensajes subjetivos, ya sea explícitamente definidos en la clase o heredados de una superclase.

> **Objeto Subjetivo**: *objeto con la habilidad de responder a un mismo mensaje de diferentes maneras, de acuerdo a determinadas circunstancias.*

Decimos que un **mensaje** es **subjetivo** si posee más de una implementación[1] asociada al mismo, y cualquiera de ellas puede llegar a ser ejecutada. Forma parte del protocolo definido para un objeto subjetivo.

Mensaje Subjetivo: *mensaje que, formando parte del protocolo de un objeto subjetivo, posee más de una implementación asociada al mismo, y cualquiera de ellas puede llegar a ser ejecutada.*

Sabemos que en un ambiente que maneja el concepto de clases, las mismas son como "fábricas" de objetos que poseen determinadas características en común. Decimos que una **clase** es **subjetiva** si los objetos a los que representa son subjetivos.

Clase Subjetiva: *clase que representa a objetos subjetivos.*

Por abuso de notación decimos que un **método** es **subjetivo** si es una de las implementaciones asociadas a un mensaje subjetivo. Hablamos de "abuso de notación" porque en realidad no es el método el que varía su implementación sino el mensaje. Una vez que para un mensaje subjetivo se determina cuál es el método a ejecutar, la forma de trabajo es análoga a la tradicional. La clave está justamente en buscar la forma de asociar un mensaje con diferentes métodos y saber cuál es el que se debe ejecutar en cada momento, o sea: generar un mecanismo de toma de decisiones que determine cuál es el método que corresponde utilizar. Este tema no es trivial, motivo por el cual se desarrollará posteriormente cuando se explique el Motor de Subjetividad.

De las definiciones anteriores se deduce que tener una clase subjetiva, no implica que todos los mensajes de la misma sean subjetivos, sino que al menos implemente o herede un mensaje que sí lo sea. De esta forma en una clase tenemos mensajes subjetivos y mensajes no subjetivos (a los que también llamamos **mensajes normales**). Esto tiene una razón de ser, y es que los objetos de una clase pueden tener mensajes que no varían su comportamiento y otros que sí. Por lo general parecería ser que los mensajes asociados a propiedades esenciales del objeto (por ejemplo: el número de una cuenta corriente) están asociados a mensajes normales. De todos modos, el programador es el que debe definir qué tipo de mensaje especificar, y así es como encaramos este tema en la tesis.

Retomemos nuestro caso de estudio y veamos cómo se aplica al mismo lo que acabamos de mencionar.

*Primero analicemos la **clase "Stack"**. Los mensajes "isEmpty" y "push" no varían su comportamiento de acuerdo con la cantidad de elementos dentro del stack. A estos mensajes son a los que denominamos "mensajes normales".*

MENSAJE	***METODO***
isEmpty	*^ (elements size == 0).*
push: anElement	*elements addLast: anElement.*

[1] La implementación de un mensaje es lo que comúnmente se conoce con el nombre de "método".

Pero no sucede lo mismo para los mensajes "pop" y "top". Estos sí reaccionan diferente de acuerdo con la cantidad de elementos existentes.
Básicamente podemos observar dos estados del stack: "Vacío" y "No Vacío".
Supongamos que factorizamos los métodos asociados a dichos mensajes de tal forma que podamos asociar las sentencias correspondientes a cada estado, y así lograr obtener dos implementaciones para cada uno de estos mensajes: una implementación para cuando el stack está "Vacío" y otra para cuando está "No Vacío".
De esta manera podríamos construir algo similar a un cuadro de doble entrada:

MENSAJE \ ESTADO	***VACIO***	***NO VACIO***
pop	*^nil.*	*^(elements last)*
top	*^nil.*	*element:= elements removeAtIndex: (elements size). ^element.*

Si suponemos por un instante que al momento de ejecución el stack puede resolver cuál es el método que debe ejecutar, es claro que las dos implementaciones de cada mensaje son susceptibles de ser ejecutadas en un instante dado.

Entonces, por definición decimos que los mensajes "pop" y "top" son subjetivos, ya que "poseen más de una implementación asociada al mismo, y cualquiera de ellas puede llegar a ser ejecutada". Como estos mensajes forman parte de la definición del comportamiento del stack, se deduce de aquí que la clase "Stack" es subjetiva, pues sus instancias poseen comportamiento subjetivo.

Encapsulamiento y Ocultamiento de Información

¿Cómo afecta la Subjetividad al "Encapsulamiento" y al "Ocultamiento de Información" ?

El "Encapsulamiento" permite tratar al objeto como una sola entidad en el sentido que la información y el comportamiento del mismo se mantienen dentro del objeto como si fuera una cápsula. Esta cápsula se ve y se manipula como una caja negra, dado que la única forma de afectar a un objeto es enviándole mensajes al mismo utilizando su interfase pública, ocultando de esta forma la estructura interna del objeto, lo que se denomina "Ocultamiento de Información". [JAC/92] [WIR/90]

La gran ventaja del "Ocultamiento de Información" es que el objeto puede cambiar su estructura interna sin modificar su interfase, con lo cual no afecta al resto de los objetos. Es útil para aumentar el nivel de abstracción. Como consecuencia se obtiene un código modificable y extensible.

A fin de lograr una verdadera extensión de los conceptos básicos de la Orientación a Objetos, la Subjetividad debe respetar ambos conceptos. Para alcanzar esto, los objetos deben encapsular los diferentes métodos asociados a un mismo mensaje subjetivo, como así también los mecanismos de decisión correspondientes para determinar cuál de estos métodos ejecutar.

Además, los objetos deben ocultar a sus clientes el hecho que su comportamiento sea subjetivo, haciendo que su interfase pública no revele su naturaleza. Esto significa que el emisor nunca sabrá qué tipo de mensaje le está enviando al receptor, ya que éste último define una única interfase, sin hacer distinción entre mensajes subjetivos y no subjetivos.

De esta forma la Subjetividad afianza los conceptos mencionados previamente, haciendo que el comportamiento subjetivo de un objeto sea transparente para los clientes del mismo, dado que no saben de qué forma les están respondiendo al servicio solicitado.

Continuando con nuestro caso de estudio, consideremos la siguiente situación:

Un objeto X le envía el mensaje "pop" a un objeto (aStack) de la clase "Stack", utilizando las siguientes sentencias de programación como parte de la implementación del mensaje "m" (el cual forma parte de su protocolo):

```
X >> m

m
| anElement aStack |
...
anElement := aStack pop.
...
```

Como puede notarse, el emisor (objeto X) invoca al mensaje "pop" sin realizar ninguna consideración particular sobre el mismo, y sin embargo el mensaje "pop" es un mensaje definido como subjetivo en la clase "Stack". Por otro lado, la clase "Stack" cuenta con los diferentes métodos[2] (met 1,met 2, ... , met N) asociados al mensaje "pop" y posee el mecanismo de decisión que le permite determinar cuál de estos métodos ejecutar.

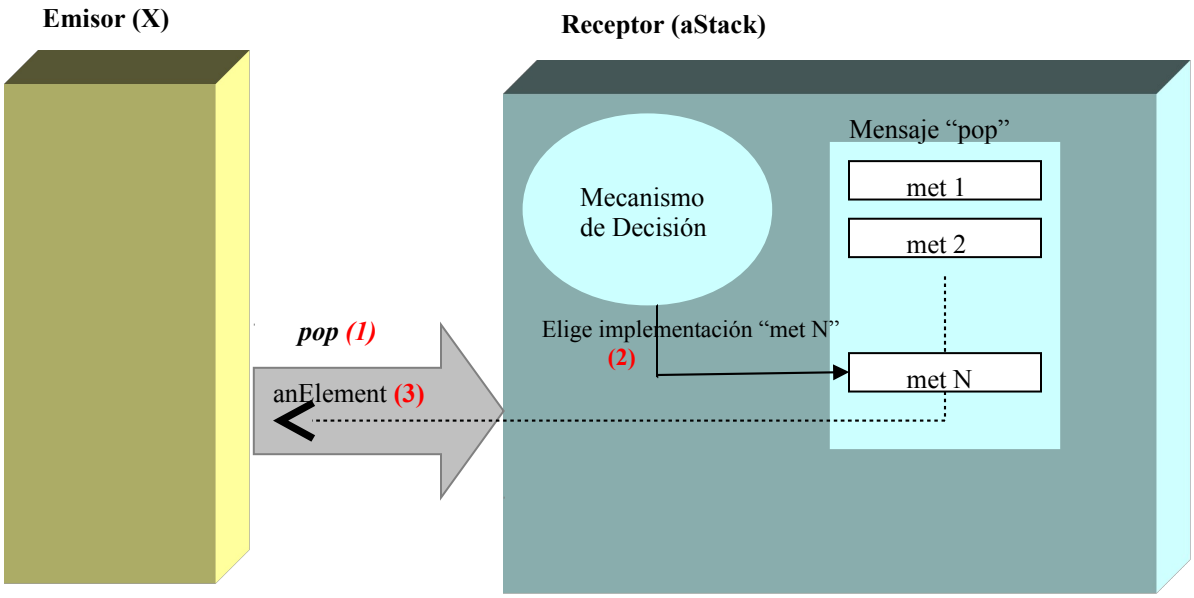


Figura 6 : Encapsulamiento y Ocultamiento del comportamiento Subjetivo

[2] Recordar que los métodos son las implementaciones asociadas a los mensajes.

Polimorfismo y Sharing de Información

El “Polimorfismo” y la “Sharing de Información” son dos conceptos básicos de la Orientación a Objetos.

El “Polimorfismo” es la habilidad por la cual objetos de diferentes clases responden a un mismo mensaje de diferente manera. Esto no significa que un objeto necesite saber a quién le es enviado el mensaje, sino que diferentes tipos de objetos responderán aquel mensaje en particular. Tampoco necesita interiorizarse sobre cómo se resuelve dicho mensaje. [WIR/90]

La intención de la programación orientada a objetos es proveer una manera natural y directa de describir los conceptos del mundo real, permitiendo la flexibilidad de expresión necesaria para capturar la naturaleza variable del mundo que debe modelarse, y la habilidad dinámica de representar situaciones cambiantes. Una parte fundamental de la naturalidad de expresión provista por la programación orientada a objetos es la habilidad de compartir datos, código y definición –a lo que denominamos “Sharing de Información”-, y en este punto todos los lenguajes orientados a objetos proveen alguna manera de definir un nuevo objeto en términos de uno ya existente, compartiendo tanto la implementación como el comportamiento del objeto definido previamente. [STE/88]
Se reconocen dos mecanismos fundamentales para implementar el “sharing” en los lenguajes de orientación a objetos: “Delegación” y “Herencia”. Históricamente, ha existido mucho debate sobre cuál de estos mecanismos era el concepto más poderoso para implementar la generalización/especialización. Desde 1987, sabemos que la delegación puede modelar la herencia, y a su vez, la herencia puede modelar la delegación. Durante “Object-Oriented Programming Systems, Languages and Applications” (OOPSLA) de 1987 llevada a cabo en Orlando, Florida (USA), Lynn Stein, Henry Lieberman y David Ungar discutieron sus diferencias entre la delegación y la herencia, y llegaron a una conclusión que reflejaba la necesidad de ambos mecanismos. Esta resolución se conoce como el “Tratado de Orlando”. [STE/88]

La “Delegación” es el mecanismo utilizado para implementar el “sharing de información” en ambientes basados en prototipos; en cambio, en ambientes basados en clases, el mecanismo utilizado para este fin es el de “Herencia”.

Debido a que nuestro trabajo se desarrolla en un ambiente basado en clases, haremos hincapié en el segundo concepto: el de “Herencia”.

La “Herencia”, como uno de los mecanismos de implementación del “sharing de información” utilizado en ambientes basados en clases, es la habilidad que tiene una clase para definir el comportamiento y la estructura de datos de sus instancias como un superconjunto de la definición de otra u otras clases. Es decir, una clase es casi como otra clase, excepto por algunas cosas más. La herencia permite pensar a una nueva clase como refinamiento de otra, abstrayendo las similitudes entre las clases y diseñando y especificando solamente las diferencias entre ambas. [WIR/90]

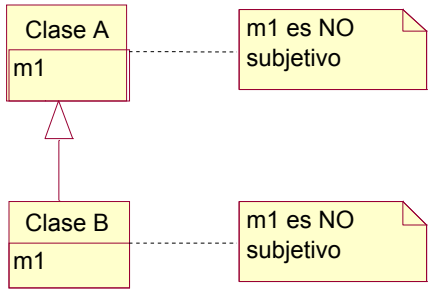
Estos conceptos también deben ser respetados por la Subjetividad, que como veremos no solamente se mantienen en los ambientes subjetivos sino que además extienden su alcance.

En primer lugar debemos realizar la siguiente distinción: la herencia de la estructura interna de un objeto no varía entre un ambiente subjetivo y uno no subjetivo. Esto ocurre pues la subjetividad está definida sobre el comportamiento de un objeto, y no sobre la información que contiene el mismo. Por lo tanto no es necesario hacer consideraciones especiales sobre la herencia en este aspecto. Simplemente debemos considerar la herencia de comportamiento.

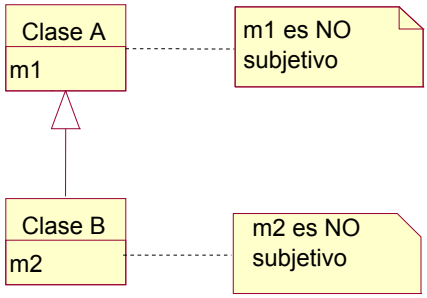
A continuación podremos observar cómo se extienden los comportamientos subjetivos y no subjetivos a través de la herencia. Para ello analicemos qué sucede en cada uno de los siguientes casos:

1. A es una clase no subjetiva, y B una subclase no subjetiva.

a)



b)

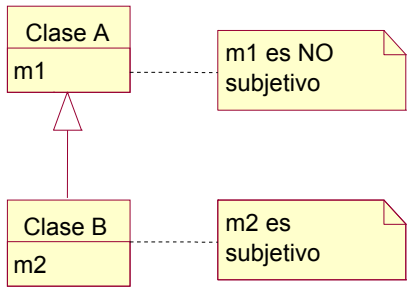


En el caso a), la clase B sobreescribe un mensaje normal definido en la superclase A, y podría eventualmente definir sus propios mensajes.
En el caso b), la clase B no sobreescribe ningún mensaje de A, aunque podría eventualmente definir sus propios mensajes.

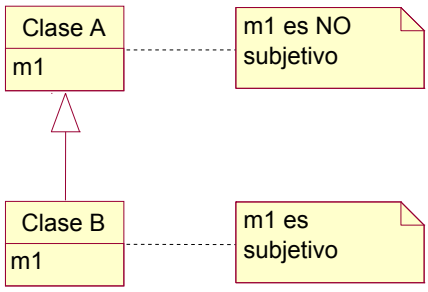
Esta es la subclasificación clásica. No se deben hacer consideraciones especiales.

2. A es una clase no subjetiva, y B una subclase subjetiva.

a)



b)



Por definición, al ser B una clase subjetiva, tiene al menos un mensaje subjetivo. Esto podría suceder por dos motivos:

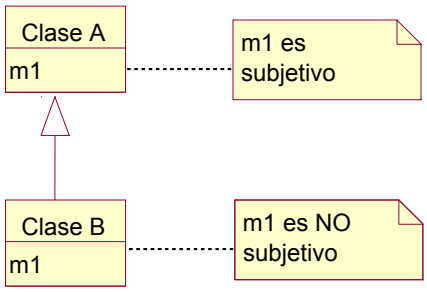
Caso a) : La clase B extiende el comportamiento definido en la clase A, agregando un nuevo mensaje del tipo subjetivo en la clase B.

Caso b): B sobreescribe como subjetivo un mensaje normal existente en A.

La experiencia actual nos indica que este es uno de los casos más frecuentes de subclasificación subjetiva, que se presenta ante la necesidad de expandir un comportamiento de manera subjetiva. Esto es así, porque si partimos de una ambiente donde aún no existe ninguna clase subjetiva, y queremos agregar una que sí lo sea, la primer subclasificación que haremos será crear una subclase subjetiva a partir de una que no lo es. Y esta situación parecería repetirse en forma bastante frecuente. Una vez que

uno cuente con un conjunto considerable de clases subjetivas, esta situación seguramente cambiará, y es probable que uno comience a subclasificar de clases subjetivas (este caso lo veremos más adelante).

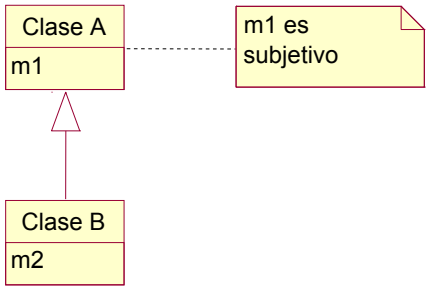
3. A es una clase subjetiva, y B una subclase no subjetiva.



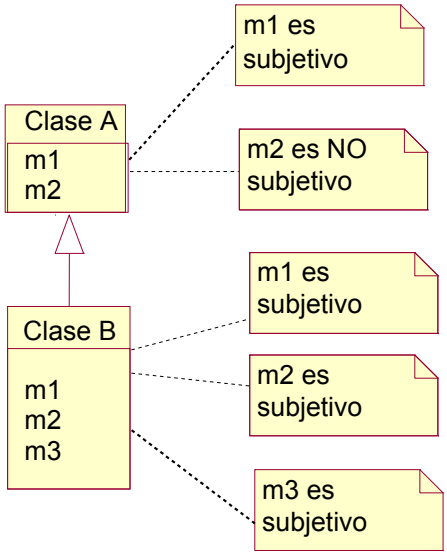
Acá primero debemos hacer una aclaración. Decir que B es una subclase no subjetiva significa (por definición) que B sobreescribió como normales todos los mensajes subjetivos de A. Subclasificar de esta manera no parecería ser una decisión de diseño razonable. Creemos que debería factorizarse el comportamiento subjetivo del no subjetivo en una superclase común a ambas. De todos modos, como es una decisión de diseño, nuestra implementación no desea restringir este caso.
En tal caso este es un tema que deberá tratarse cuando se analicen y definan las metodologías de diseño para trabajar con el nuevo ambiente.

4. A es una clase subjetiva, y B una subclase subjetiva.

a)



b)



Caso a) : La clase B hereda el comportamiento subjetivo de A, sin extenderlo ni modificarlo. Seguramente extenderá el comportamiento no subjetivo de A.

Caso b): Acá puede pasar que B extienda el comportamiento de A de alguna de las siguientes formas:

- sobreescribiendo mensajes subjetivos con otros mensajes también subjetivos.
- sobreescribiendo mensajes no subjetivos como subjetivos.
- agregando nuevos mensajes subjetivos.

Notemos que todas estas situaciones pueden llegar a ser decisiones de diseño totalmente razonables.

¿Como afecta la subjetividad al Polimorfirsmo?

En los ambientes no subjetivos el polimorfismo ocurre entre dos objetos dados, debido a que responden a un *mismo mensaje* de maneras semánticamente equivalentes. Dos clases son polimórficas si sus instancias también lo son. De este modo, las dos clases definen el mismo mensaje y este mensaje en ambas clases resulta implementado por métodos polimórficos: tienen el mismo nombre, los mismo tipos de parámetros, los mismos efectos colaterales, el mismo tipo de resultado y el mismo propósito, aunque no tienen los mismos detalles de implementación. [WOO/97]
De aquí se desprende que dos objetos polimórficos pueden ser intercambiados en forma transparente para el emisor del mensaje debido a que tienen una implementación distinta del mensaje, pero que es semánticamente equivalente. En forma trivial, un objeto es polimórfico consigo mismo, ya que estamos refiriéndonos a la misma implementación del mensaje.

En ambientes subjetivos se logra que un objeto sea polimórfico consigo mismo de manera no trivial, respondiendo a un *mismo mensaje* de *diferentes maneras* , siendo éstas semánticamente equivalentes. Esto sucede porque un mismo mensaje se implementa con métodos distintos, los cuales son polimórficos.

Concluímos entonces que en ambientes subjetivos el polimorfismo es extendido, permitiendo que un objeto sea polimórfico consigo mismo de manera no trivial.

*Continuando con nuestro ejemplo, analicemos ahora la clase **"LimitedStack"**.*

Por tratarse de un stack limitado en cuanto a la cantidad máxima de elementos que puede almacenar, podemos observar que los estados posibles de este tipo de stack son: "Vacio", "Intermedio" y "Lleno".

La clase "LimitedStack" hereda de la clase "Stack" los métodos "pop" y "top", y no los sobreescribe como normales, con lo cual, de acuerdo con nuestra definición ("...en su interfase contienen uno o más mensajes subjetivos, ya sea explícitamente definidos en la clase o heredados de una superclase."), la clase "LimitedStack" es subjetiva porque hereda comportamiento subjetivo. Notar que esta condición alcanza para decir que la clase "LimitedStack" es subjetiva, más allá de que defina sus propios métodos subjetivos. De todos modos sobreescribe el método normal del mensaje "push" definido en la clase "Stack", con un método subjetivo para considerar el caso del stack lleno. O sea: además de poseer comportamiento subjetivo heredado, agrega más comportamiento subjetivo definiéndolo explícitamente en su propia clase.

MENSAJE \ ESTADO	***VACIO***	***INTERMEDIO***	***LLENO***
pop (*)			
top (*)			
push	*(**)*	*(**)*	*^nil.*

(*) Estos mensajes están implementados en la superclase "Stack".

(**) Estos métodos no se implementan para estos determinantes, con el objetivo de recurrir al mecanismo de herencia y utilizar los de la superclase. (Este tema se ampliará en la sección de Lógica de Fuerzas)

Desde el punto de vista del Polimorfismo, cuando un objeto envía el mensaje "push" a una instacia de la clase "LimitedStack", la respuesta obtenida corresponde a la ejecución de alguno de los métodos implementados para el mensaje en cuestión (por ejemplo, si el stack está vacío se responde con un método distinto a cuando el stack está lleno). Esto significa, que el envío de un mensaje "push" a un mismo objeto de la clase "LimitedStack" en diferentes momentos, puede dar como resultado la ejecución de distintos métodos, los cuales tienen el mismo nombre (push) , los mismo tipos de parámetros (anElement, un elemento para el Stack) , los mismos efectos colaterales (el stack en un estado definido), y el mismo tipo de resultado (en este caso, no retornan nada), el mismo propósito (insertar un elemento en el stack, si es posible), aunque no tienen los mismos detalles de implementación (se especifican en las diferentes celdas de la tabla anterior).

El Motor de Subjetividad

Las Fuerzas

Como dijimos anteriormente, para nosotros la **Subjetividad** se define como *"la habilidad que tiene un objeto de comportarse de diferentes maneras ante una situación de acuerdo a los estímulos o factores (internos o externos) que lo influyen en un momento determinado."*

Estos estímulos o factores pueden pensarse como **fuerzas** que se ejercen sobre los objetos, afectando su comportamiento.

La Fuerza de Estado

Recordemos el ejemplo de la cuenta bancaria. La misma puede estar en diferentes estados: habilitada (para operatoria normal), inhabilitada (por falta de fondos, por orden judicial, etc.), o cerrada. Dependiendo de estos estados, una misma operación realizada sobre la cuenta tendrá resultados diferentes. En este caso, el estado de la cuenta afecta el comportamiento de las operaciones (por ejemplo una extracción de dinero).

Algo similar sucede con la interfase gráfica (GUI), que según el estado en el que se encuentra (abierta, cerrada, etc.), las operaciones realizadas sobre la misma pueden diferir su comportamiento. [TAI/93]

Lo mismo puede observarse con el Stack de nuestro caso de estudio. El estado del stack (determinado por la cantidad de elementos del stack) afecta su comportamiento.

Como se puede observar en los ejemplos mencionados previamente, el estado interno de un objeto puede llegar a afectar su comportamiento, haciendo que responda de diferente manera a un mismo mensaje. En los ambientes no subjetivos, estos casos normalmente se resuelven construyendo sentencias condicionales del tipo "if-case", para chequear su estado, y en base a ello decidir qué rama del método ejecutar, asegurando la ejecución de la porción de comportamiento adecuado para una situación en particular.

Para estos casos en los cuales el estado interno influye en el comportamiento de los objetos, proponemos la utilización de una fuerza que denominamos "**Fuerza de Estado**".

Al referirnos al estado interno de un objeto, lo hacemos en el sentido de considerar un determinado estado lógico del objeto, de la manera en que se lo define en [TAI/93]. Las variables de instancia de un objeto pueden ayudar a determinar cuál es el estado en el que se encuentra dicho objeto, pero no es una condición necesaria que exista una variable de instancia que determine dicho estado en cada momento.

En un momento específico, un objeto puede encontrarse en cierto estado lógico y luego, por diversos motivos, puede pasar a otro estado lógico. Para utilizar la Fuerza de Estado es necesario definir cuáles son todos los estados posibles en los que se puede encontrar el objeto. Denominaremos *transición de estados* al pasaje de un estado interno a otro. Estas transiciones de estados pueden llegar a resultar complejas, por lo cual nos parece conveniente la utilización de *Diagramas de Transiciones* o *Autómatas* para modelarlas [AHO/90]. Los motivos por los cuales un objeto puede sufrir una transición de estado pueden estar asociados a la simple recepción de un mensaje o al cambio de valores de un conjunto determinado de variables de instancia .

En el ejemplo del LimitedStack, los estados lógicos del objeto son: Vacío, Intermedio y Lleno. Las transiciones de estado se pueden modelar con el siguiente autómata:

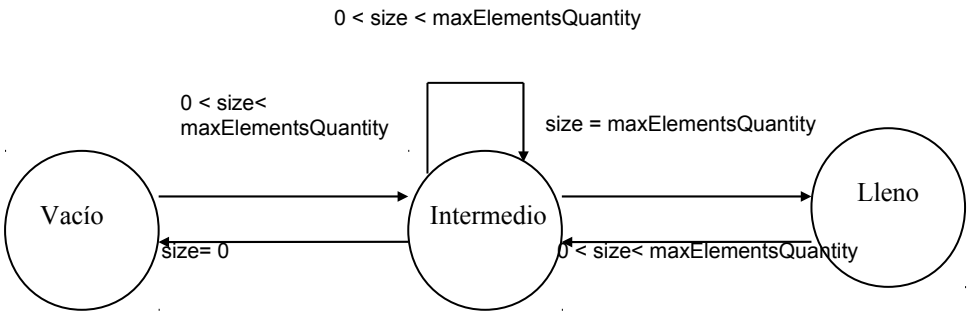


Figura 7: Autómata para las transacciones de estado del LimitedStack

El mecanismo de decisión deberá contar con la lógica necesaria para manejar el pasaje de transaciones, de forma similar a la que se modela en este autómata.

La utilización de autómatas permite poder manejar situaciones de error, ya que toda transición no especificada pasaría a un "estado de error". El diseñador debe considerar cómo se comporta el objeto según el estado en el que se encuentra; esto le permite concentrarse en la programación de un estado lógico a la vez. Como consecuencia, se pueden reestructurar las sentencias condicionales que dependan del estado interno del objeto, distribuyéndolas a través de los métodos asociados a cada estado.

Hemos encontrado que existen similitudes entre la utilización de la Fuerza de Estado y el Pattern State. Veamos el ejemplo de la Clase TCPConection, explicado en la sección del Pattern State en el Apéndice I (y proveniente originalmente de [GAM/95]), utilizando la Fuerza de Estado.

El diagrama de clases es simplemente el siguiente:

TCPconnection

ActiveOpen()
PasiveOpen()
Close()
Acknowledge()
Send()
Synchronize()

Los estados lógicos definidos para la Fuerza de Estado son los que representan los diferentes modos en los que se pueda encontrar la conexión:

- Established
- Closed
- Listening

El mecanismo de decisión implementará las transiciones de estado mediante el siguiente diagrama de transiciones:

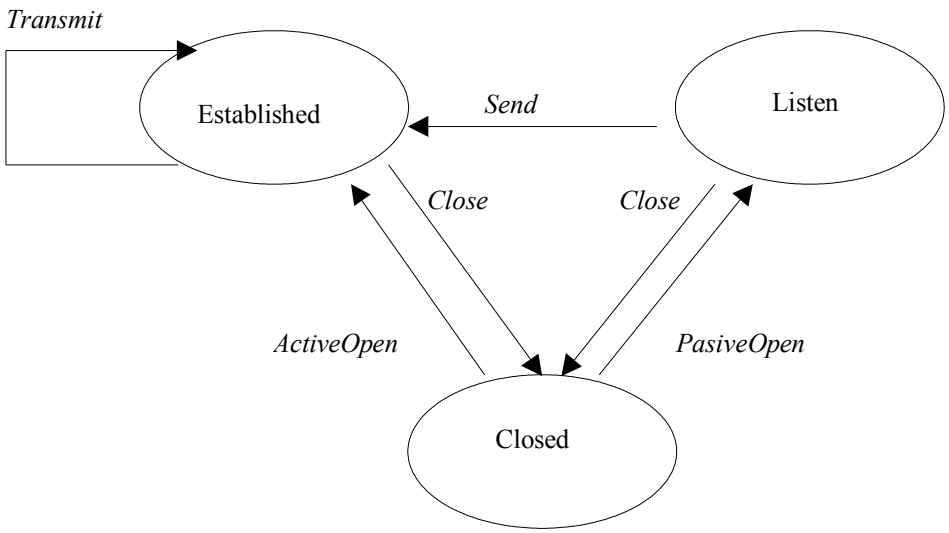


Figura 8: Autómata para las transacciones de estado de TCPConnection

A continuación comentamos en forma detallada las similitudes y diferencias que hemos encontrado entre esta Fuerza y el Pattern State:

- El propósito de este Pattern y la Fuerza de Estado son equivalentes, es decir: ambos pueden resolver el mismo tipo de problema: que un objeto varíe su comportamiento de acuerdo con su estado interno.

- La Fuerza de Estado es aplicable en los mismos casos que el Pattern State: tanto cuando el comportamiento de un objeto depende de su estado interno y debe variar en tiempo de ejecución dependiendo de este estado, como cuando se quieren re-estructurar grandes sentencias condicionales que dependen del estado del objeto.

- Con la Fuerza de Estado se mantienen las mismas ventajas, pero no se incrementa el número de clases como en el Pattern State:

  - Al utilizar el Pattern State se debe generar una clase por cada estado. Esto hace que se incrementen el número de clases a medida que aumentan los estados. En cambio, al utilizar la fuerza de estado no ocurre este problema ya que los diferentes estados lógicos se definen dentro de la fuerza y están dentro de la misma clase cuyo comportamiento se quiere variar.

  - Los estados se pueden compartir mediante el mecanismo habitual de "sharing": la herencia[3]. Las transiciones de estado pueden ser definidas mediante un autómata, como se explicó

[3] Recordar que la herencia es el mecanismo que permite implementar "sharing" en un ambiente basado en clases, que es justamente el tipo de ambiente sobre el cual estamos trabajando.

anteriormente. Esto evita que las subclases de State tengan que conocerse entre sí y depender una de otra.

- El comportamiento específico de cada estado está localizado en los métodos asociados a ese estado.

En un principio, el uso de la "Fuerza de Estado" parece obtener resultados semejantes a los obtenidos con el uso del Pattern State, sin el agravante del aumento en la cantidad de clases.

La Fuerza del Emisor

Otro de los factores que afecta el comportamiento de un objeto es el emisor del mensaje.

En la "Introducción" de nuestro trabajo mencionamos que las personas responden de modo diferente a una misma pregunta dependiendo de quién la esté realizando.

Lo mismo sucede cuando un objeto debe representarse a sí mismo como una descripción textual si una instancia de TextPane está solicitando su representación. Pero si el objeto solicitante es una instancia de la clase GraphicPane, entonces su representación debe ser una imagen. Más aún, si una instancia de la clase Printer está realizando el pedido entonces la respuesta podría ser una combinación de ambas. Notar que el mensaje que se envía permanece constante mientras que la respuesta es la que varía, aunque todas las respuestas sean representaciones del receptor. [PRI/97]
Podemos ver que el comportamiento de este objeto variará dependiendo de quién sea el emisor: un TextPane, un GraphicPane o una Printer.

Para estos casos en los cuales el emisor del mensaje influye el comportamiento de los objetos, proponemos la utilización de una fuerza que denominamos "**Fuerza del Emisor**".

A diferencia de la Fuerza de Estado, no se cuenta con una herramienta como un diagrama de Transiciones para modelar la Fuerza del Emisor. La similitud es que el diseñador deberá conocer los diferentes "emisores" de mensajes que hacen que el objeto varíe su comportamiento.

Estos emisores conocidos pueden definirse por comprensión o por extensión.

En el primer caso, simplemente se puede conocer la clase a la cual pertenecen los objetos emisores de mensajes. Bastará con que el mecanismo de decisión conozca el conjunto de clases para las cuales tiene que tener un comportamiento variable y determine, según la clase del emisor , a cuál de esos conjuntos corresponde el comportamiento que debe ejecutarse.

En el segundo, se determina explícitamente cada objeto emisor de mensaje. Estos objetos pueden guardarse en diferentes colecciones, según el comportamiento al que correspondan. Cuando se recibe un mensaje, el mecanismo de decisión deberá determinar a qué colección pertenece el emisor del mensaje y de esa manera conocerá el comportamiento que debe ejecutarse.

Siguiendo con el ejemplo del objeto que debe representarse de diferentes formas, este objeto podría guardar una colección de emisores conocidos por comprensión. En este caso se tendrá una colección como la siguiente:

(TextPane, GraphicPane, Printer)

Si aparece alguna otra clase que solicite una representación del objeto, habrá que agregarla a esta colección.

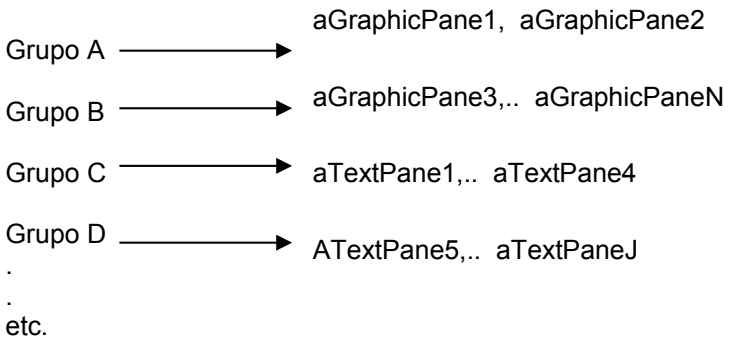
Si se optara por la otra alternativa, la de definir los objetos emisores conocidos por comprensión, habría que mantener una colección con estos objetos. Aquí, la colección sería la siguiente:

(aTextPane1, aTextPane2, ..aTextPaneN, aGraphicPane1,..aGraphicPaneM, aPrinter1, ..aPrinterK)

Como vemos, este último caso puede llegar a tener que manejar colecciones mucho más grandes.

Una tercer alternativa puede ser definir un esquema intermedio que permita agrupar a los objetos emisores conocidos por alguna característica en común, y definir el comportamiento variable según estos grupos.

En este caso se tendrían dos niveles de colecciones: una primer colección para definir los grupos de emisores y luego, para cada uno de los grupos, una colección para definir los objetos asociados a estos grupos.



Como vemos, la Fuerza del Emisor debe proveer al diseñador diferentes alternativas para que pueda definir el comportamiento variable según los emisores del mensaje. Estos emisores conocidos deben estar identificados ya sea por comprensión o por extensión.

La Fuerza del Contexto

En el mundo real las personas juegan diferentes roles cuando están inmersas en diferentes contextos. Digamos que la situación externa los fuerza a tomar diferentes roles. Por ejemplo, un profesor de la universidad cuando está enseñando se comporta más formalmente que cuando está conversando en una reunión de amigos, aunque en ambos casos la operación básica es la misma, y hasta podría estar conversando sobre el mismo tema. El contexto que rodea a la persona lo fuerza a que se comporte de manera diferente.

En general, las situaciones o contextos en los cuales está participando un objeto también puede afectar el comportamiento del mismo. Un contexto puede tener diferentes significados para diferentes objetos dependiendo del dominio de la aplicación, la tecnología de implementación, etc. Más aún, algunos objetos serán afectados por algunos contextos mientras que para otros ese contexto puede no tener

efecto. Esto puede verse fácilmente si necesitamos que un mismo objeto responda de diferentes modos a un mismo mensaje de acuerdo a la aplicación donde está participando.

Otro caso donde es posible observar cómo influye el contexto, puede ser en un hospital que se encuentra en alerta de emergencia. En esta situación la conducta completa del hospital está alterada. Los casos son tratados con diferentes prioridades, los insumos y los recursos del hospital son asignados con otro criterio, y también está afectado el reclutamiento de personal.

Para estos casos en los cuales el contexto influye el comportamiento de los objetos, proponemos la utilización de una fuerza que denominamos "**Fuerza del Contexto**".

Para poder utilizar esta fuerza, se debe proveer al diseñador la posibilidad de identificar los contextos conocidos para los cuales varía el comportamiento. Consideramos que las alternativas aquí son similares a la Fuerza del Emisor, es decir: se pueden definir los contextos conocidos en colecciones, tanto por extensión como por comprensión.

La Fuerza del Contenido

Originalmente las fuerzas detectadas eran: Estado, Emisor y Contexto. Pero desde un inicio sabíamos que seguramente éstas no eran todas las fuerzas existentes. Era posible que nuevas fuerzas aparecieran en la medida que comprendiéramos cómo se comportan los objetos del mundo real. Por este motivo intentamos plantear nuestro trabajo desde un punto de vista más general, considerando la existencia de N fuerzas influyendo sobre los objetos. El tiempo terminó dándonos la razón: durante el desarrollo nos encontramos con una nueva fuerza, a la que denominamos "Fuerza del Contenido" (o "Content Force").

Bajo el esquema presentado por esta fuerza, cada objeto decide cómo comportarse de acuerdo con los parámetros recibidos en la invocación de un mensaje. Esta fuerza parece aplicar fundamentalmente a colecciones de objetos, si pensamos en colecciones inteligentes que deciden comportarse en forma distinta de acuerdo con los objetos que almacenan. También podrían necesitar un comportamiento especial en el momento de retornar los objetos almacenados, de acuerdo con el tratamiento que hayan recibido al guardarlos.
El mecanismo de decisión puede manejar a "los contenidos" por comprensión o por extensión, dándole un tratamiento similar al utilizado para la fuerza del emisor.

Un ejemplo de uso de esta fuerza podría ser una OrderedCollection que maneje un algoritmo de ordenamiento diferente, dependiendo del tipo de contenido que guarde. En este caso se deberá tener una colección paralela que defina los tipos de contenidos, al estilo de la utilizada por la Fuerza del emisor al definir los emisores por comprensión. La implementación del método subjetivo definirá el algoritmo de ordenamiento según el tipo de contenido. Notar que en este caso puede también utilizarse el Pattern Strategy (ver Apéndice I).

Aplicando las Fuerzas

Como primera medida, resumamos las fuerzas que acabamos de explicar:

- **Fuerza de Estado (o "State Force"):** cada objeto decide cómo comportarse de acuerdo con su estado interno.

- **Fuerza del Emisor (o "Sender Force"):** cada objeto decide cómo comportarse de acuerdo con el emisor del mensaje.

- **Fuerza de Contexto (o "Context Force"):** cada objeto decide cómo comportarse de acuerdo con el contexto (ambiente) en el que está involucrado.

- **Fuerza del Contenido (o "Content Force"):** cada objeto decide cómo comportarse de acuerdo con los parámetros recibidos en la invocación de un mensaje.

Veamos cómo se aplican estos conceptos a nuestro caso de estudio.

Fuerza de Estado: Para el caso de la clase "Stack" analizado hasta ahora, considerar la cantidad de elementos existentes para realizar una u otra operación, es tener en cuenta su estado interno. De este modo, cuando definimos las distintas implementaciones anteriores, lo estábamos haciendo para la "Fuerza de Estado".

Fuerza del Emisor: Supongamos ahora que consideramos una nueva restricción para la clase "Stack": solamente podrán extraer elementos del stack determinados objetos. Por ejemplo: podríamos decir que solamente podrán extraer elementos aquellos usuarios que se encuentren autorizados. En este caso, al tener en cuenta quiénes son los emisores del mensaje, estamos teniendo en cuenta la "Fuerza del Emisor". Cabe aclarar que para determinar quiénes son los usuarios autorizados, se podría realizar un chequeo por extensión ("usuario A", "usuario B", etc.) o por comprensión ("todos los objetos de la clase XX", o "todos los objetos que cumplan determinada propiedad...").
Otro ejemplo donde puede observarse cómo influye esta fuerza sobre un stack, es cómo se representa el objeto a sí mismo: puede hacerlo como una descripción textual si el que está solicitando su representación es una instancia de TextPane, o como una imagen si el que lo está haciendo es una instancia de GraphicPane.

Fuerza del Contenido: Supongamos que un stack desea agregar inteligencia para ahorrar espacio de almacenamiento, y de este modo decidir compactar o no un determinado objeto a almacenar, de acuerdo a su tipo, haciéndolo transparente a sus emisores. Aquí ya no importa quién esté invocando el mensaje "push", sino que lo que importa es el tipo de elemento que se intenta guardar, o sea: el parámetro "anElement". En este caso, estamos teniendo en cuenta la "Fuerza del Contenido".

Observando la realidad está claro que estas fuerzas coexisten en todo momento, y debe determinarse qué método utilizar para un mensaje dado bajo determinadas circunstancias, o sea: bajo la fuerza influyente en dicho momento sobre el objeto. [PRI/97].
Originalmente se pensaba que existía cierta precedencia predeterminada entre las fuerzas (en forma fija), pero llegamos a la conclusión que llevar a cabo una implementación de este estilo era muy restrictiva. En realidad es el desarrollador el que tiene que poder especificar la fuerza predominante en cada momento de acuerdo a los objetos que esté modelando. Por eso debe desarrollarse un mecanismo para tal fin, al que de aquí en adelante denominaremos **lógica de fuerzas**.

Los Determinantes

Cada una de estas fuerzas posee lo que denominamos **determinantes**. Estos representan diferentes instancias de estudio dentro de cada fuerza.

Por ejemplo, para la fuerza "Estado", los determinantes son diferentes estados lógicos en los cuales se puede encontrar un objeto. Para la fuerza "Emisor", los determinantes son los diferentes emisores de mensajes (clases o conjuntos de objetos); para la fuerza de contexto, los determinantes son los distintos contextos en los que actúa el objeto; para la fuerza de contenido, los determinantes son los distintos "tipos" de parámetros que espera recibir un objeto.

Si bien las fuerzas son las mismas para todas las clases subjetivas existentes dentro del ambiente, no sucede lo mismo con los determinantes de cada una en diferentes clases. Notemos que los

determinantes de la fuerza “Estado” para la clase Stack –por ejemplo: “Vacío”/”No Vacío” – son diferentes a los determinantes de la fuerza “Estado” para la clase CuentaBancaria –por ejemplo: “Con Fondo” / “En Rojo”, etc.-.

De esta forma el comportamiento subjetivo de un objeto se definirá en función de la fuerza y el determinante.

Retomemos el ejemplo del “LimitedStack”: el comportamiento subjetivo del Stack varía de acuerdo a si se encuentra “Lleno”, “Vacío” o “Intermedio”. Si consideramos que estos estados surgen a partir de analizar el estado interno del objeto (cantidad de elementos en el stack), podríamos decir que esta clase está regida por la fuerza de “Estado”, y los determinantes resultan ser “Lleno”, “Vacío”, e”Intermedio” (y todos los estados del stack que se deseen tener en cuenta). De esta manera, tendremos un comportamiento para la fuerza “Estado”, determinante “Lleno”, otro comportamiento para la fuerza “Estado”, determinante “Vacío”, y otro para la fuerza “Estado”, determinante “Intermedio”.

Desde el punto de vista metodológico, es razonable pensar que los determinantes definidos en una clase, para una determinada fuerza, deben ser disjuntos, y la unión de ellos debe representar un todo.

Por una cuestión de simplicidad, de aquí en adelante cada vez que hablemos de “fuerza” predominante, en realidad nos estaremos refiriendo a “fuerza y determinante”.

Planteada la idea que un objeto reaccione en forma diferente ante el envío de un mismo mensaje de acuerdo a la fuerza predominante, aparece la necesidad de que el objeto disponga de distintas implementaciones para un mismo mensaje. Aquí surgen claramente dos cuestiones:

a. ¿ Cómo determinamos la fuerza predominante sobre un objeto en un momento determinado?
b. ¿ Cómo resolvemos la ejecución del método correspondiente de acuerdo a la fuerza obtenida?

Con el objetivo de comprender mejor estos dos temas, es que desarrollaremos los siguientes temas: “La Lógica de Fuerzas” y “El Mecanismo de Despacho de Mensajes”.

La Lógica de Fuerzas

Denominamos **Lógica de Fuerzas** al mecanismo de decisión que determina cuál es la fuerza que prevalece sobre un objeto subjetivo en un instante dado, y como consecuencia de ello, se determina cuál es la implementación del mensaje (método) que será ejecutada.

La Lógica de Fuerzas especifica qué aspectos debe tener en cuenta un objeto para tomar dicha decisión. Tales aspectos pueden ser su estado interno, el emisor del mensaje, el tipo de parámetro, y otros.

Este mecanismo debe:

- estar presente en todo objeto subjetivo para que pueda decidir cuál es la fuerza predominante al momento de recibir un mensaje.

- activarse ante el envío de un mensaje subjetivo al objeto en cuestión, ayudando a que éste realice la decisión adecuada de qué método utilizar para responder al mensaje recibido. En un caso no subjetivo, cuando un objeto recibe un mensaje, simplemente se ejecuta el único método asociado al mismo. Pero con Subjetividad tenemos más de un método para un mensaje, por lo cual se debe recurrir a la Lógica de Fuerzas para determinar cuál de todos los métodos definidos debe ejecutarse.

¿Cómo podríamos definir la lógica de fuerzas para nuestro caso de estudio?

Veámoslo para la clase "Stack". En este caso, la fuerza que influye el comportamiento del objeto, es la "Fuerza de Estado" debido a que se tiene en cuenta la cantidad de elementos del mismo, lo cual forma parte de su estado interno. Recordemos los determinantes definidos para esta fuerza: "Vacío" y "No Vacío".
La lógica de fuerzas deberá responder entonces que la fuerza predominante es la "Fuerza de Estado" y el determinante es "Vacío" si el stack no tiene elementos. Caso contrario, responderá con "Fuerza de Estado" y el determinante "No Vacío". De esta manera, si asociáramos un método a cada fuerza y determinante, sabríamos cuál es el método que corresponde ejecutar.

Es razonable pensar que cada clase posee su propia Lógica de Fuerzas. Pero en el momento de ejecución no es cierto que todos los objetos de una misma clase sean afectados por la misma fuerza. Cada objeto posee su propia identidad, con lo cual los contenidos de sus variables de instancia pueden diferir, los emisores de los mensajes también, etc.

De aquí se deduce la necesidad de evaluar la lógica de fuerzas sobre cada objeto en sí mismo, más allá de que pertenezcan a una misma clase. Y más aún, esta evaluación debe efectuarse ante cada envío de mensaje que se le realice a dicho objeto, dado que al variar los determinantes (su estado interno, los emisores del mensaje, etc.), la fuerza predominante sobre el mismo puede ser distinta, determinando cuál es el método que debe ejecutarse.

En un principio pensamos que alcanzaba con definir la lógica de fuerza para la clase del objeto, o sea: sin tener en cuenta el mensaje que se está recibiendo. Pero en realidad esto puede no ser así, y suceder que la decisión de la fuerza predominante tuviese que realizar consideraciones diferentes para distintos mensajes. Para comprender esto un poco mejor, veamos el siguiente ejemplo:

Hasta este punto, tal cual hemos visto en nuestro caso de estudio, la fuerza que influye el comportamiento de los objetos de la clase "Stack" es la fuerza "Estado". Ahora bien: ¿Qué sucedería si tuviéramos que contemplar un "push" subjetivo con las siguientes características: si el objeto a apilar es una imagen de tamaño mayor de 2 MB[4], compactar el objeto antes de apilarlo; sino proceder simplemente a apilarlo, sin realizar otra operación?[5]
En este caso, la fuerza predominante es "Contenido". Y sin embargo para el resto de los mensajes la fuerza sería "Estado". ¿Qué sucede entonces?

Esto nos indica que la lógica de fuerzas puede tener que realizar consideraciones diferentes de acuerdo con el mensaje que se recibe, surgiendo la necesidad de poder definir la lógica de fuerzas para un mensaje, además de definirla para la clase.

¿Entonces... Lógica de Fuerzas se define para la clase o para un mensaje?

Habría que permitir ambas posibilidades.

Esto significa que si se puede definir una lógica de fuerzas para la clase que no discrime el mensaje recibido, debe poder definirse esta lógica una sola vez. Esto le evitaría al programador definir la misma lógica de fuerzas para todos los mensajes que posean una lógica de fuerzas en común. En cambio, si es necesario considerar el mensaje recibido, debe poder definirse esta lógica para dicho mensaje.

[4] Nótese que esta condición es simplemente a modo de ejemplo; en realidad la condición puede ser genérica.
[5] Observar que en caso de implementar el "push" aquí descripto, el método "pop" deberá contemplar la descompactación automática del objeto desapilado, para que la inteligencia planteada en el stack sea transparente a los clientes del mismo, y al desapilar un objeto del stack los clientes reciban el mismo objeto que apilaron.

Como podría darse el caso que la lógica de fuerzas difiera solamente para un mensaje (o unos pocos) y no para todo el resto, sería cómodo contar con la opción de definir tanto una lógica para la clase como una lógica para un mensaje, en forma simultánea.

Pero aquí surge la siguiente pregunta...

¿Qué lógica de fuerza se tiene en cuenta en tiempo de ejecución?

Una opción es buscar primero una lógica de fuerza asociada al mensaje que se está recibiendo; si se encuentra definida, entonces se utiliza dicha lógica; sino se busca la lógica de fuerzas definida para la clase.
Esta metodología permite factorizar la lógica de fuerzas común a varios mensajes y definirla a nivel de la clase, siendo la precedencia a nivel de mensaje y en caso de no existir esta definición evaluar la especificada a nivel de clase.

Cabe aclarar que es un error de diseño definir un mensaje subjetivo y no definir lógica de fuerza ni a nivel de mensaje ni a nivel de clase.

Esquemáticamente, para buscar la lógica de fuerzas en tiempo de ejecución se puede proceder de la siguiente manera:

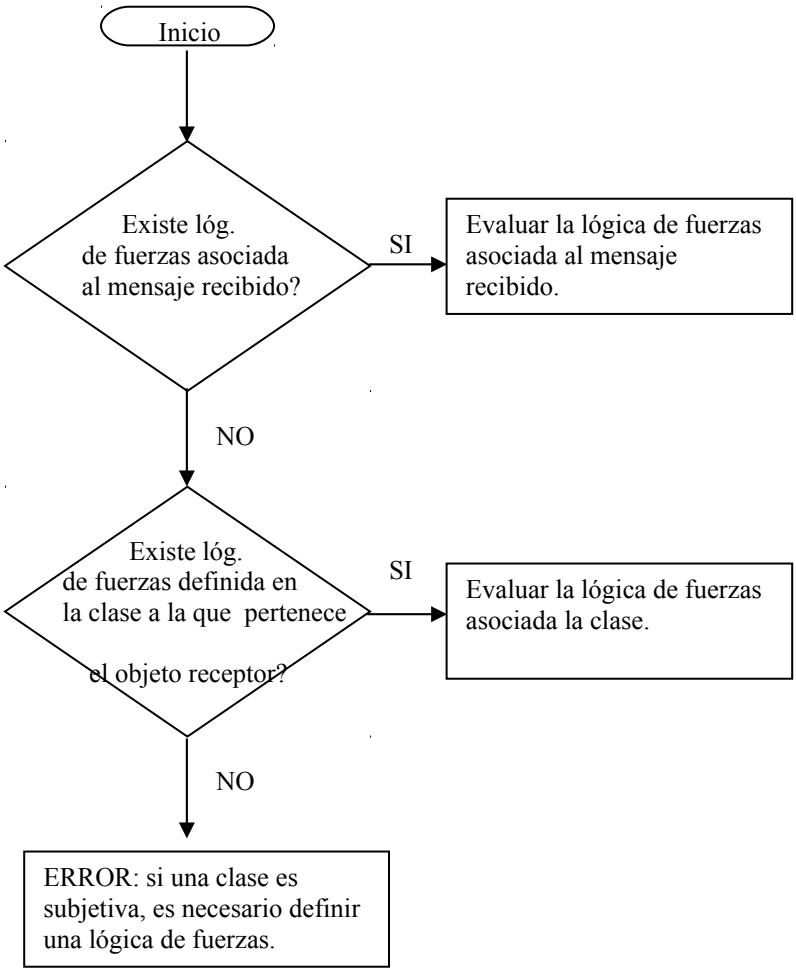


Figura 9: Esquema de búsqueda de lógica de fuerzas

*Veámoslo más detenidamente con un ejemplo. Analicemos cómo debe ser la lógica de fuerzas para cada uno de los mensajes de nuestra **clase "Stack":***

- *Mensajes "isEmpty" y "initialize": debido a que estos mensajes no son subjetivos, no se realizan consideraciones especiales.*

- *Mensaje "push" (según como lo acabamos de definir): para este mensaje se debe considerar si el objeto a apilar es una imagen mayor a 2 MB para compactarla; en caso afirmativo la fuerza será "Contenido" y el determinante será "Compactable"; en caso negativo, la fuerza será "Contenido" y el determinante será "No compactable".*

- *Mensajes "pop" y "top": para ambos mensajes se debe realizar la misma consideración: "Verificar si el stack se encuentra vacío"; en caso afirmativo, la fuerza predominante será "Estado" y el determinante será "Vacío"; de lo contrario, habrá que verificar si el elemento que se encuentra en el tope del stack fue previamente compactado; de ser así, la fuerza predominante será "Contenido" y el determinante "Compactable"; de otro modo, la fuerza será la misma y el determinante "No Compactable".*

Debido a que los mensajes "pop" y "top" deben realizar las mismas consideraciones, podemos definir la lógica de fuerzas a nivel de la clase, y para el mensaje "push" definirla a nivel de mensaje.

Sintéticamente... para la clase "Stack":

MENSAJE \ CONDICION	***OBJETO A APILAR > 2 MB***	***OBJETO A APILAR < 2 MB***
push *(lógica de fuerzas definida a nivel de método)*	*Fuerza = "Contenido"* *Determinante = "Compactable"*	*Fuerza = "Contenido"* *Determinante = "No Compactable"*

MENSAJE \ CONDICION	***STACK VACIO***	***STACK NO VACIO***	
		OBJETO COMPACTADO	***OBJETO NO COMPACTADO***
pop *top* *(lógica de fuerzas definida a nivel de clase)*	*Fuerza = "Estado"* *Determinante = "Vacío"*	*Fuerza = "Contenido"* *Determinante = "Compactable"*	*Fuerza = "Contenido"* *Determinante = "No Compactable"*

¿Qué ocurre cuando un objeto de la clase "Stack" recibe cada uno de estos mensajes?

- *Mensajes "isEmpty" y "initialize": como estos mensajes forman parte del comportamiento no subjetivo del stack, no se evaluará lógica de fuerzas alguna (a pesar que esté definida a nivel de clase), ya que tienen un único método asociado.*

- *Mensaje "push": evaluará la lógica de fuerzas definida a nivel del mensaje.*

- *Mensajes "pop" y "top": como los mismos no tienen definida lógica de fuerzas a nivel de mensaje, evaluarán la definida a nivel de clase.*

Resumiendo, la lógica de fuerzas decide cuál es la fuerza y el determinante que predominan al momento de recibir un mensaje subjetivo, para saber en base a esto qué método debe ejecutarse. Pero...

¿ Qué ocurre si no existe un método asociado a esa fuerza y determinante ?

En este caso se recurre al mecanismo de herencia: como no existe una implementación del mensaje en la clase en cuestión, recurrimos al mecanismo de herencia para utilizar el comportamiento de dicho mensaje definido en la superclase, que puede ser subjetivo o no. En caso de no serlo, se ejecuta el único método que implementa el mensaje. En caso de serlo, se evalúa nuevamente la lógica de fuerzas definida en la superclase, pudiendo resultar que la fuerza predominante sea distinta a la que predominó en la subclase. Cómo se ve esto en un ejemplo?

Retomemos el ejemplo del "LimitedStack", y veamos cuál sería la lógica de fuerzas asociada:

MENSAJE \ CONDICION	***LIMITEDSTACK VACIO***	***LIMITEDSTACK CON ELEMENTOS***	***LIMITEDSTACK LLENO***
push	*Fuerza = "Estado"* *Determinante = "Vacio"*	*Fuerza = "Estado"* *Determinante = "Intermedio"*	*Fuerza = "Estado"* *Determinante = "Lleno"*

El único método que a "LimitedStack" le interesa sobreescribir es el método "push" bajo la condición de limitedStack lleno, para rechazar el apilado del elemento en cuestión. Por lo tanto solamente existirá implementación bajo la fuerza "Estado" y determinante "Lleno". Para el resto de los casos, no definirá método.

Al recibir el mensaje "push", si el limitedStack se encuentra lleno ejecutará el método asociado a la fuerza "Estado" y determinante "Lleno". En caso contrario, se recurrirá al mecanismo de herencia, ya que como dijimos anteriormente, la clase "LimitedStack" no redefine el método "push" en esos casos.
Al recurrir al mecanismo de herencia, en la superclase (Stack) se reevaluará la lógica de fuerzas para el mensaje "push", la cual decidirá que la fuerza predominante será "Contenido" y el determinante podrá ser "Compactable" o "No Compactable" de acuerdo con el objeto que se desea apilar.

De esta forma podemos observar que, al recurrir al mecanismo de herencia, la fuerza predominante resultó distinta a la que se había determinado al momento de evaluar la lógica de fuerzas de "LimitedStack".

¿ Por qué en caso de utilizar el mecanismo de herencia debe reevaluarse la lógica de fuerzas a nivel de la superclase?

Porque lo que uno está heredando es el comportamiento subjetivo tal cual está definido en la superclase. Esto significa que heredar el mensaje subjetivo implica heredar todos los métodos asociados al mismo, y también el mecanismo de decisión (lógica de fuerzas) necesario para determinar cuál de estos métodos ejecutar.

Si esto no fuera así, no se estaría manteniendo el concepto de herencia de comportamiento.

De lo dicho en el párrafo anterior, deducimos que la lógica de fuerzas se hereda, pero no como una unidad independiente, sino que está incluída en el mecanismo de despacho del mensaje subjetivo que se está heredando. Esto significa que no puede ocurrir que a un mensaje le falte una lógica de fuerzas dentro de la clase donde está definido, y que se le intente asociar la lógica de fuerzas definida en la superclase.

Hasta aquí hemos hablado de herencia de lógica de fuerzas y de métodos; veamos qué ocurre con la herencia de los determinantes.

Cuando para una clase se desea utilizar los mismos determinantes que fueron definidos en su superclase, se deben definir en ésta, ya que no existe herencia de determinantes en forma directa, pero sí en forma indirecta en tiempo de ejecución. Esto es: no se puede hacer referencia a un determinante definido únicamente en la superclase desde la lógica de fuerzas de la subclase, ni tampoco definir que un método es la implementación de un mensaje para una fuerza y dicho determinante. El primer caso dará error en tiempo de ejecución, y el segundo mientras se programa el método.

Decimos que los determinantes se heredan indirectamente, pues en el caso de recurrir al mecanismo de herencia, ya sea porque el mensaje no tiene implementación en la subclase, o porque no se encuentra definido el método asociado a la fuerza predominante sobre el objeto, se trabajará con los determinantes definidos en la superclase, ya que la lógica de fuerzas de la superclase trabaja con dichos determinantes.

Por lo tanto, los determinantes se heredan en tiempo de ejecución en el caso de recurrir al mecanismo de herencia.

Veamos este concepto en el siguiente ejemplo, recordemos que los determinantes para la fuerza de Estado en la clase "Stack" son "Vacío" y "No Vacío", y en la clase "LimitedStack" son "Vacío", Ìntermedio" y "Lleno".
Como "LimitedStack" sobreescribe el método "push:", se deben definir explícitamente los tres determinantes. En cambio, si volvemos al ejemplo del push que compacta al objeto a apilar, "LimitedStack" hereda completamente la implementación de Stack para dicho mensaje; esta herencia incluye la lógica de fuerzas, los métodos que implementan dicho mensaje, y también incluye a los determinantes definidos para la fuerza Content : "Compactable" y "No Compactable", ya que la lógica de fuerzas definida en "Stack" hace referencia a los mismos.

El Mecanismo de Despacho de Mensajes

Uno de los puntos clave del trabajo se centra en la resolución del mecanismo de definición y despacho de mensajes (subjetivos y no subjetivos). Esto implica poder distinguir con qué tipo de método se está trabajando y actuar en función a ello.

Sin lugar a dudas la mayor dificultad aparece con los mensajes subjetivos. ¿Cómo asociamos a un mismo mensaje subjetivo distintas implementaciones para el mismo? ¿Y cómo sabemos qué método ejecutar cuando se realiza la invocación a dicho mensaje? Y esto es más interesante aún si mantenemos la idea de que no proliferen los mensajes, o sea: que de alguna forma todas las implementaciones realizadas para un determinado mensaje estén asociadas a *dicho* mensaje. Esta es una forma de obtener objetos polimórficos consigo mismos de manera no trivial.

La idea es realizar el esquema de un despacho multi-método. Decimos que es multi-método pues recibido un mensaje, se debe optar por una de las varias implementaciones del mismo, a diferencia de los ambientes no subjetivos donde, por cada mensaje existe solamente un método que lo implementa.

Cuando un objeto recibe un mensaje, verifica si dicho mensaje es subjetivo o no. En caso de no ser subjetivo, ejecuta el método asociado al mismo. En caso de ser subjetivo, debe decidir cuál de todas las implementaciones del mismo se debe ejecutar, debiendo realizar dicha elección en base a la fuerza predominante sobre el objeto en ese instante.

Para ello recurre a la lógica de fuerzas y en base al resultado retornado, se obtiene y ejecuta el método correspondiente. En caso de no estar definido el método se procede a utilizar el mecanismo de herencia, repitiendo el procedimiento descripto, pero esta vez en la superclase.

Notar que todo este mecanismo es transparente para el objeto emisor del mensaje.

Retomemos el ejemplo del "Stack", y recordemos las distintas implementaciones para los mensajes "pop" y "top".

MENSAJE \ ESTADO	***VACIO***	***NO VACIO***
pop	*^nil.*	*element := elements removeAtIndex: (elements size). ^element .*
top	*^nil.*	*^(elements last).*

Veamos ahora qué sucede con el mecanismo de despacho de mensajes cuando un objeto de esta clase recibe el mensaje "pop".

Lo primero que se debe hacer es determinar cuál es la fuerza predominante en ese instante. Para ello recurre a la lógica de fuerzas, la que analiza la el estado del stack en base a la cantidad de elementos en el mismo. Si el stack no se encuentra vacío, la fuerza predominante es "Estado" y el determinante es "No Vacío".

A continuación se procede a ejecutar el método definido para esta fuerza y determinante.

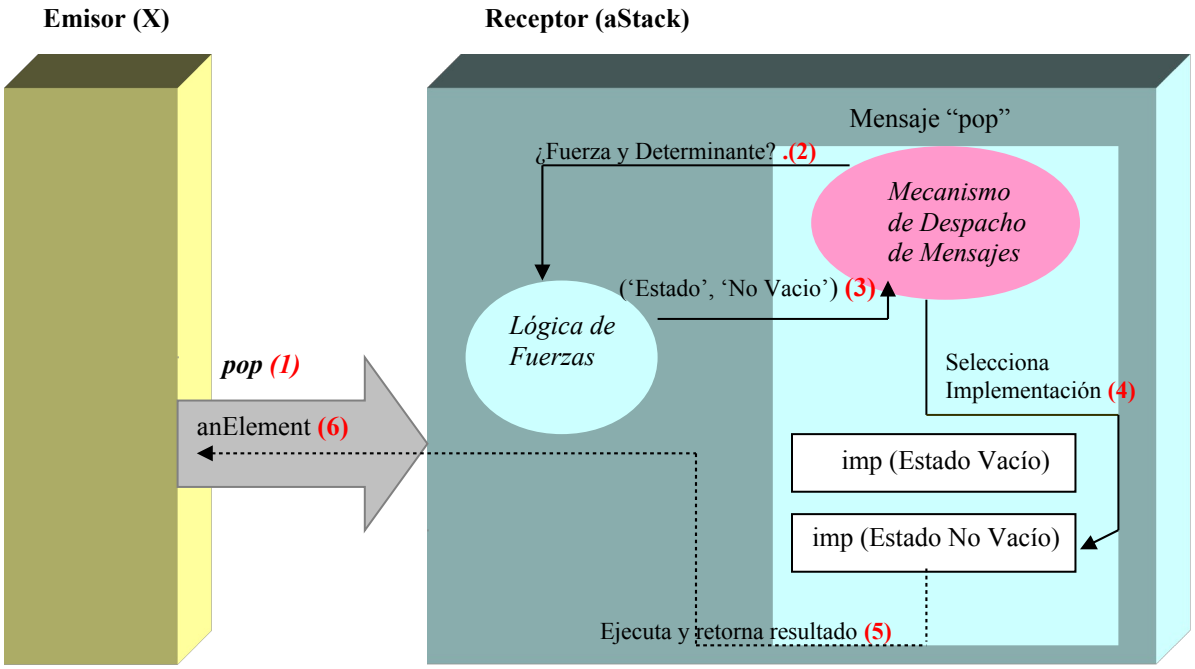


Figura 10 : Mecanismo de Despacho de Mensajes

Herramientas de desarrollo en un ambiente subjetivo

El Browser

Hasta este punto hemos hablado del tipo de extensiones a realizar para permitir el manejo de subjetividad, pero aún no hemos abordado el tema de las modificaciones que deben realizarse a las herramientas del ambiente actual de tal forma que encajen dentro del nuevo esquema de trabajo.

La primer herramienta que claramente debe modificarse es el browser. Debe extenderse su funcionalidad de tal forma que le permita al programador realizar todas las operaciones actuales, y además brindarle un acceso claro y amigable a la información subjetiva de cada clase permitiéndole, por ejemplo, especificar qué método utilizar con cada fuerza y determinante. Básicamente debe tener la siguiente funcionalidad:

- Alta, Baja, Modificación y Consulta de las fuerzas existentes en el ambiente.
- Alta, Baja y Consulta de los determinantes de una fuerza para una clase determinada.
- Alta, Baja, Modificación y Consulta de métodos normales. En realidad esto no debe considerarse como una extensión en sí, sino como operaciones que deben mantenerse para contar con un ambiente mixto. La clave está en distinguir qué tipos de métodos se están agregando.
- Alta, Baja, Modificación y Consulta de métodos subjetivos; de acuerdo con la implementación a realizar debe considerarse especialmente la compilación y el almacenado de estos métodos.
- Alta, Modificación y Consulta de la lógica de fuerzas de cada clase.

Si bien en las próximas secciones veremos este tema en más detalle, veamos gráficamente cómo podría verse el browser propuesto; para ello observemos la próxima figura:

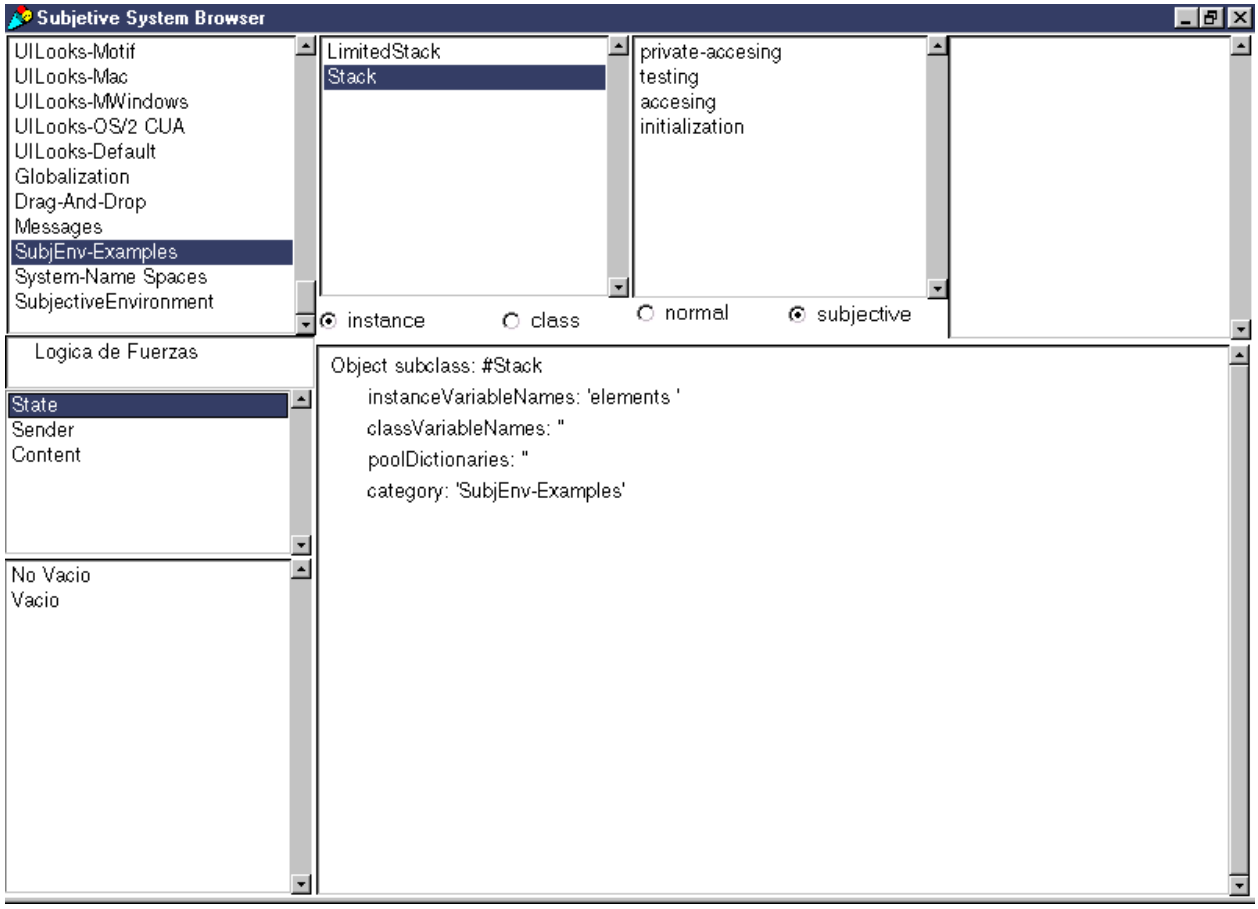


Figura 11: Browser propuesto para un ambiente subjetivo.

Los Inspectors

Un Inspector es una herramienta que permite examinar y eventualmente modificar un objeto. Permite tener una vista de la estructura interna (variables de instancia) de un objeto en un momento dado, tal como si fuera una foto instantánea del interior del mismo, con la posibilidad de realizarle modificaciones.

En general los inspectors se visualizan a través de ventanas que tienen dos partes:

a) Una lista de las variables de instancia del objeto examinado.
b) El contenido de la variable de instancia seleccionada de la lista a).

En un ambiente subjetivo esta funcionalidad debería poder extenderse a:

- Visualización y modificación de la fuerza y determinante que influye sobre el objeto inspeccionado.
- Visualización y modificación de las diferentes fuerzas y determinantes definidos para el objeto inspeccionado (incluyendo los objetos definidos por extensión sobre los cuales se desea que el objeto inspeccionado se comporte de distintas maneras).

El Debugger

Otra herramienta que debe extenderse es el debugguer, proporcionando al programador los datos necesarios para que pueda realizar un seguimiento de las sucesivas invocaciones de mensajes, con la posibilidad de apreciar el comportamiento subjetivo (si es que lo hay) y la lógica de fuerzas; de esta forma podrá saber cómo está respondiendo un objeto y ver cómo afectan las fuerzas al comportamiento del objeto en tiempo de ejecución.

La funcionalidad adicional debería ser:

- Obtención de fuerza y determinante de un método subjetivo (o sea: la fuerza y determinante bajo las cuales el programador definió el método).
- Modificación del código de un método normal y reiniciado de la ejecución desde dicho punto.
- Modificación del código de un método subjetivo y reiniciado de la ejecución desde dicho punto.
- Modificación de la lógica de fuerzas y reiniciado de la ejecución desde dicho punto.

Si bien en las próximas secciones veremos este tema en más detalle, veamos gráficamente cómo podría verse el debugger propuesto; para ello observemos la próxima figura:

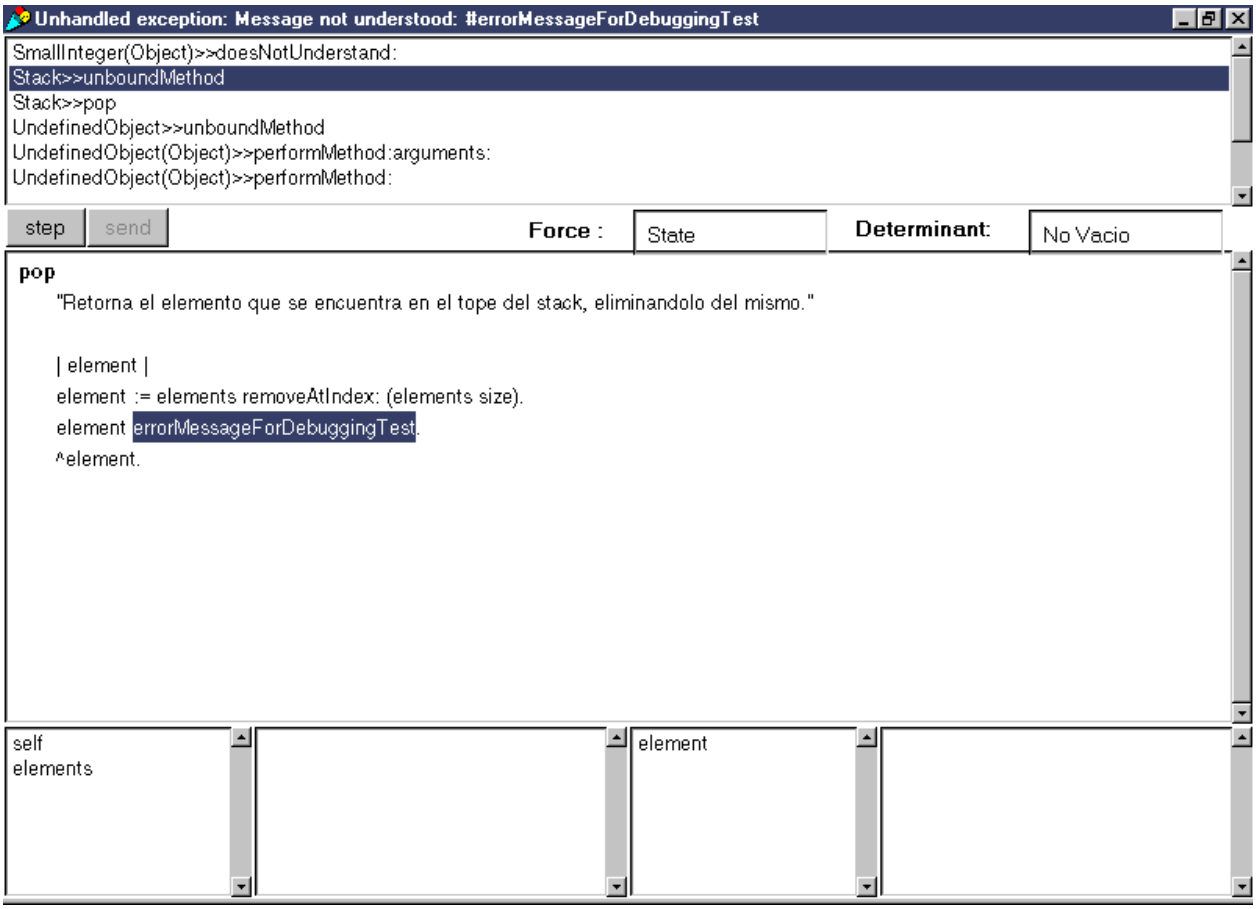


Figura 12: Debugger propuesto para un ambiente subjetivo.

El Administrador de Fuerzas

A esta altura del trabajo es evidente que las fuerzas que influyen a los objetos son un factor fundamental. Como mencionáramos en puntos previos, el ambiente a implementar no debería limitarse a un número predeterminado de fuerzas, ya que no estamos exentos de la aparición de nuevas fuerzas a lo largo del tiempo. Con lo cual sería interesante contar con una herramienta nueva –a la que de aquí en más denominaremos **Administrador de Fuerzas** – que permita realizar el agregado, eliminación o modificación de fuerzas en el ambiente de trabajo, de tal forma que el programador pueda realizar estas operaciones en forma fácil y amigable.

Características de las extensiones a realizar

Hasta ahora hemos mencionado las extensiones que deben realizarse a las herramientas del ambiente subjetivo que necesitamos desarrollar. Pero no hemos mencionado aún las características que deben poseer estas extensiones para evitar que se conviertan en cambios tan invasivos que compliquen a los desarrolladores:

- Transparencia: Las extensiones realizadas deben ser transparentes para los programadores que utilicen el ambiente. Esto significa que:
  - la sintaxis del lenguaje debe mantenerse, de tal forma que el código pueda seguir escribiéndose de manera tradicional.
  - si un programador no desea o no necesita definir comportamiento subjetivo, puede seguir trabajando como lo hacía con el ambiente no subjetivo tradicional.
- Compatibilidad: Extender el ambiente para que permita manejar Subjetividad, no debe implicar la necesidad de recompilar o reescribir código ya existente.

Las Clases involucradas en nuestra solución

En primer lugar describiremos cada una de las clases involucradas en nuestra solución, y sus correspondientes responsabilidades. Para ello utilizaremos el formato de tarjetas CRC.

CLASE	***SubjectivityBehavior*** **(concreta)**
Descripción : Representa la información necesaria para manejar el ambiente subjetivo: fuerzas existentes y definiciones de lógica de fuerzas. La variable global “SubjectivitySmalltalk” es una instancia de esta clase.	
Superclase	Object
Subclases	-
Responsabilidades	**Colaboraciones**
Devolver una fuerza con un nombre dado	Force OrderedCollection
Devolver la lista de nombres de todas las fuerzas existentes	Force OrderedCollection
Devolver el organizador de una fuerza (para saber cuáles son los determinantes definidos para cada clase)	Force OrderedCollection ClassOrganizer
Devolver la lista de determinantes para una fuerza y una clase	Force ClassOrganizer
Indicar si existe una fuerza con un nombre determinado	Force OrderedCollection
Agregar una fuerza con un nombre dado a la colección de fuerzas	Force OrderedCollection
Eliminar una fuerza de la colección de fuerzas	OrderedCollection
Almacenar un bloque de lógica de fuerzas para una clase	ForceLogicDefinition
Almacenar un bloque de lógica de fuerzas para una clase y un mensaje	ForceLogicDefinition
Eliminar todos los bloques de lógica de fuerzas para una clase	ForceLogicDefinition
Eliminar el bloque de lógica de fuerzas de una clase y un mensaje	ForceLogicDefinition
Devolver la fuerza y el determinante que predominan en un contexto de ejecución	ForceLogicDefinition
Devolver el bloque de lógica de fuerzas para una clase	ForceLogicDefinition
Devolver el bloque de lógica de fuerzas para una clase y un mensaje	ForceLogicDefinition
Inicializarse	OrderedCollection ForceLogicDefinition

CLASE	***Force*** **(concreta)**
Descripción : Representa a las fuerzas existentes en el ambiente subjetivo.	
Superclase	Object
Subclases	-
Responsabilidades	**Colaboraciones**
Asignar y devolver el nombre de la fuerza	
Asignar y devolver el organizador que contiene los determinantes de la fuerza para todas las clases/mensajes en que se los haya definido	
Inicializarse	String ClassOrganizer

CLASE	***InfluenceForce*** **(concreta)**
Descripción : Representa a la fuerza que influye sobre un objeto en un momento determinado.	
Superclase	Object
Subclases	-
Responsabilidades	**Colaboraciones**
Asignar y devolver la fuerza influyente	
Asignar y devolver el determinante influyente	

CLASE	***SubjectiveMethod*** **(abstracta)**
Descripción : Representa a los métodos compilados que se intercalan en los casos de mensajes subjetivos	
Superclase	CompiledMethod
Subclases	SubjectivityCompiledMethod NullCompiledMethod
Responsabilidades	**Colaboraciones**
Crear un método subjetivo a partir de una clase y del nombre del mensaje con los parámetros	Class CompiledMethod
Devolver el código para derivar la ejecución al método compilado asociado a la fuerza predominante	Debe ser implementado por la subclase
Almacenar como propios los bytecodes a partir de un método compilado	CompiledMethod
Inicializarse a partir de una clase, del nombre del mensaje con los parámetros y de un método compilado	Debe ser implementado por la subclase
Cambiar el método compilado existente dentro del diccionario de métodos de una clase por sí mismo	Class

CLASE	***SubjectivityCompiledMethod* (concreta)**
Descripción : Representa a los métodos compilados que se intercalan en los casos de mensajes subjetivos, ya que poseen la lógica para despachar el método asociado a la fuerza predominante. Almacenan las distintas implementaciones para un mismo mensaje subjetivo.	
Superclase	SubjectiveMethod
Subclases	-
Responsabilidades	**Colaboraciones**
Devolver el código predeterminado para derivar la ejecución al método compilado asociado a la fuerza predominante	Text
Inicializarse a partir de una clase, del nombre del mensaje con los parámetros y de un método compilado	Class Dictionary Collection
Devolver el método compilado que corresponde ejecutar según la fuerza y el determinante predominantes	InfluenceForce SubjectiveBehavior ExecutionContext
Agregar un método compilado bajo una fuerza y un determinante	Dictionary
Devolver el método compilado bajo una fuerza y un determinante	Dictionary
Cambiar el nombre de una fuerza en el diccionario de fuerzas	Dictionary
Agregar métodos compilados nulos bajo una fuerza en base al nombre del mensaje y de los parámetros	List Class NullCompiledMethod Dictionary
Devolver la fuerza y el determinante bajo el cual se encuentra definido un método (para debugging)	Dictionary InfluenceForce
Indicar si tiene métodos definidos	Dictionary
Indicar si incluye un método	Dictionary
Eliminar un determinante bajo una fuerza	Dictionary

CLASE	***NullCompiledMethod* (concreta)**
Descripción : Representa a los métodos compilados que se intercalan en los casos de mensajes subjetivos, ya que posee la lógica para invocar al mismo mensaje que esta implementando, pero que se encuentra definido en la superclase del objeto receptor. (Es el elemento neutro dentro de los métodos compilados. Básicamente constituye un método que simula un método compilado no definido, y ejecuta un ^super para pasar el control a la superclase que corresponda).	
Superclase	SubjectiveMethod
Subclases	-
Responsabilidades	**Colaboraciones**
Devolver el código predeterminado para derivar la ejecución al método compilado asociado a la fuerza predominante	Text
Inicializarse a partir de una clase, del nombre del mensaje con los parámetros y de un método compilado	
Crear un método subjetivo a partir de una clase y del nombre del mensaje con los parámetros	Class SubjectiveMethod

CLASE	***ExecutionContext*** **(concreta)**
Descripción : Representa la información necesaria para manejar el despacho de mensajes en un ambiente subjetivo.	
Superclase	Object
Subclases	-
Responsabilidades	**Colaboraciones**
Asignar y devolver la clase a la que pertenece el objeto receptor del mensaje	
Asignar y devolver el objeto receptor del mensaje	
Asignar y devolver el objeto emisor del mensaje	
Asignar y devolver los argumentos involucrados en el mensaje subjetivo	
Asignar y devolver el selector del mensaje	

CLASE	***ForceLogicDefinition*** **(concreta)**
Descripción : Representa a las definiciones de lógicas de fuerzas. Almacena las definiciones de lógicas de fuerzas para las clases y los mensajes definidas en el ambiente subjetivo.	
Superclase	Object
Subclases	-
Responsabilidades	**Colaboraciones**
Agregar un bloque de lógica de fuerzas para una clase y un selector (en caso de selector nulo, se asume que el alcance del bloque es a nivel de clase)	Dictionary
Devolver el bloque de lógica de fuerzas definido para una clase y un selector (en caso de selector nulo, se asume que el alcance del bloque es a nivel de clase)	Dictionary
Devolver la fuerza y el determinante que influyen sobre un objeto subjetivo de acuerdo a un contexto de ejecución	Dictionary ExecutionContext Compiler CompiledBlock Class
Devolver el selector asociado a un bloque de lógica de fuerzas en una clase	Dictionary
Eliminar todos los bloques de lógica de fuerzas definidos para una clase	Dictionary
Eliminar el bloque de lógica de fuerzas definido para una clase y un selector (en caso de selector nulo, se asume que el alcance del bloque es a nivel de clase)	Dictionary
Inicializarse	Dictionary

CLASE	***ForceAdministrator*** **(concreta)**
Descripción : Representa al administrador de fuerzas; permite crear, eliminar y cambiar el nombre de las fuerzas existentes en el ambiente.	
Superclase	SimpleListEditor
Subclases	-
Responsabilidades	**Colaboraciones**
Agregar una fuerza en el ambiente subjetivo	SubjectivityBehavior Dialog
Cambiar el nombre de una fuerza en el ambiente subjetivo	SubjectivityBehavior Force ClassOrganizer Class Dialog
Eliminar una fuerza en el ambiente subjetivo	SubjectivityBehavior Force ClassOrganizer Class Dictionary ClassOrganizer
Inicializarse	SubjectivityBehavior

CLASE	***SubjectivityBrowser*** **(concreta)**
Descripción : Representa al browser del ambiente subjetivo; permite trabajar con métodos normales, subjetivos y con lógica de fuerzas.	
Superclase	Browser
Subclases	-
Responsabilidades	**Colaboraciones**
Agregar, eliminar y modificar las fuerzas definidas en el ambiente subjetivo	ForceAdministrator SubjectivityBehavior
Agregar, eliminar y obtener los determinantes para una fuerza	Dialog ClassOrganizer
Desplegar el código fuente para un método (subjetivo o no) de una clase	SubjectivityBehavior Class SubjectivityCompiledMethod CompiledMethod
Desplegar el código del bloque de lógica de fuerzas	SubjectivityBehavior
Aceptar y compilar el código fuente de un método normal	Class Compiler
Aceptar y compilar el código fuente de un método subjetivo	Class Compiler SubjectiveMethod SubjectivityCompiledMethod CompiledMethod
Aceptar y compilar el código del bloque de lógica de fuerzas	Compiler SubjectiveBehavior

CLASE	***SubjectivityDebugger*** **(concreta)**
Descripción : Representa al debugger del ambiente subjetivo; permite seguir los métodos normales, subjetivos y lógica de fuerzas durante una ejecución.	
Superclase	Debugger
Subclases	-
Responsabilidades	**Colaboraciones**
Desplegar la fuerza y el determinante que prevalecieron sobre el método que se está inspeccionando	SubjectivityCompiledMethod
Aceptar, compilar y reiniciar la ejecución de un método normal o subjetivo	Class Compiler SubjectivityCompiledMethod CompiledMethod
Aceptar, compilar y reiniciar la ejecución del bloque de lógica de fuerzas	Class Compiler SubjectivityBehavior
Inhabilitar la modificación del código de un SubjectiveMetod (SubjectivityCompiledMethod o NullCompiledMethod)	SubjectiveMethod SubjectivityCompiledMethod NullCompiledMethod

CLASE	***SubjectivityVisualLauncher*** **(concreta)**
Descripción : Representa al launcher del ambiente subjetivo. Realiza la apertura de browsers y launchers correspondientes al ambiente subjetivo.	
Superclase	VisualLauncher
Subclases	-
Responsabilidades	**Colaboraciones**
Agregar, eliminar y modificar las fuerzas definidas en el ambiente subjetivo	ForceAdministrator SubjectivityBehavior
Abrir los browsers del ambiente subjetivo	SubjectivityBrowser
Abrir un launcher subjetivo	

CLASE	***NotifierView*** **(concreta)**
Descripción : Es la clase estándar de Smalltalk. Se modificó para que trabaje con el debugger subjetivo.	
Superclase	View
Subclases	-
Responsabilidades	**Colaboraciones**
Devolver la clase del debugger	SubjectivityDebugger

A continuación se muestra la forma en que colaboran estas clases entre sí. Las colaboraciones se muestran divididas en dos diagramas para simplificar la comprensión del mismo.

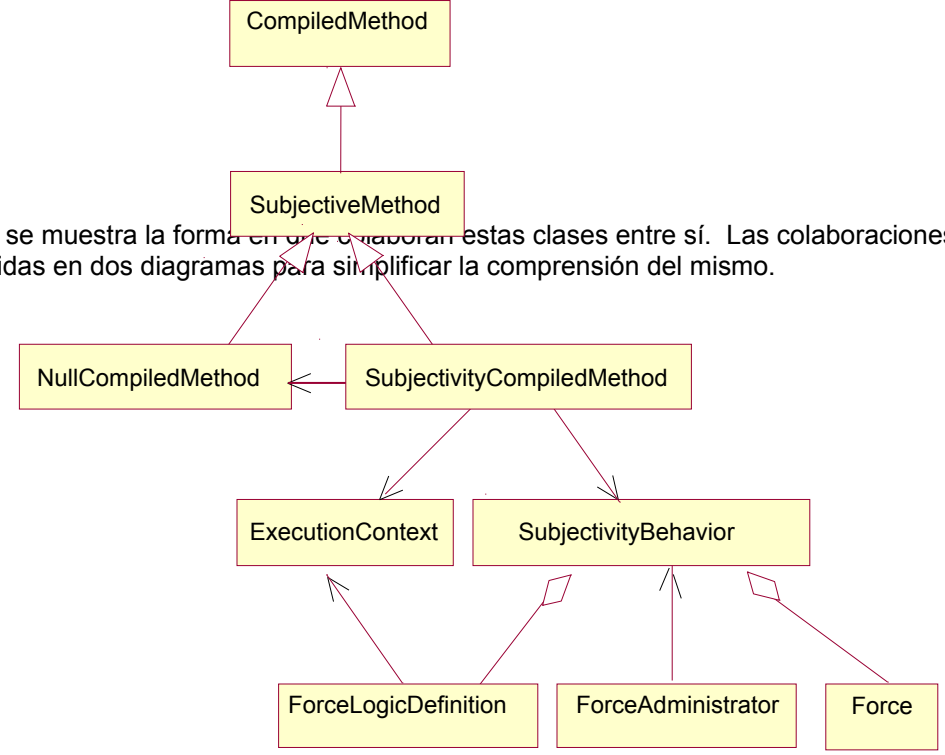


Figura 13: Diagrama de Clases de la Implementación del Ambiente Subjetivo – Motor de Subjetividad

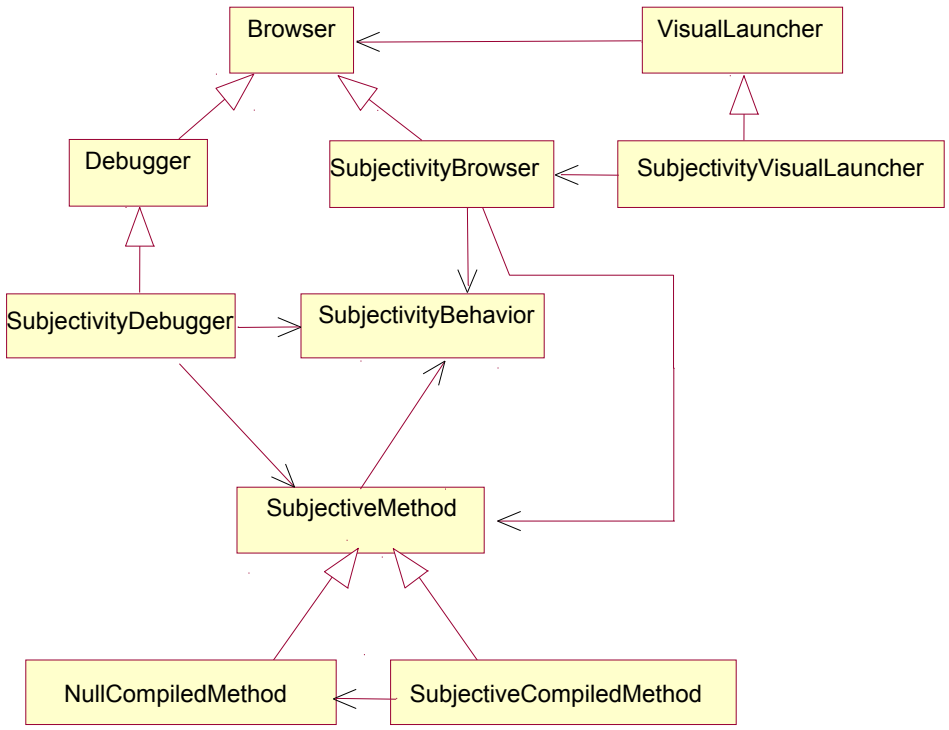


Figura 14 : Diagrama de Clases de la Implementación del Ambiente Subjetivo – Herramientas de Desarrollo

El Motor de Subjetividad

<u>El entorno del ambiente subjetivo (Clase “SubjectivityBehavior”; Instancia “SubjectivitySmalltalk”)</u>

La clase “SubjectivityBehavior” representa la información necesaria para manejar el ambiente subjetivo: fuerzas existentes en el ambiente y definiciones de lógicas de fuerzas definidas para cada clase/método. Durante uno de los pasos de instalación del ambiente subjetivo, es necesario crear una variable global (denominada “SubjectivitySmalltalk”) que es un objeto perteneciente a esta clase. Esto se lleva a cabo ejecutando:

SubjectivitySmalltalk := SubjectivityBehavior new initialize.

Esta clase tiene definida dos variables de instancia:

- forceCollection: es una colección de fuerzas, o sea: de objetos de la clase “Force”. Esta colección de fuerzas contiene todas las fuerzas definidas en el ambiente subjetivo.
- forceLogicDefinition: es una instancia de la clase “ForceLogicDefinition”, y es la encargada de almacenar los bloques de lógica de fuerzas a nivel de clase/método, en forma de relaciones del tipo:

  <clase, bloque de lógica de fuerzas> ó
  <mensaje, bloque de lógica de fuerzas>

(más adelante, cuando analicemos la clase “ForceLogicDefinition” explicaremos realmente cómo se almacenan los bloques de lógica de fuerzas).
Originalmente, la idea fue modificar la clase “Class”, pero algunas cuestiones técnicas nos llevaron a desistir de esta idea debido a que el ambiente se tornaba inestable, y además necesitábamos recompilar clases existentes, algo que preferíamos no llevar a cabo.

(Todo esto se puede ver realizando un *SubjectivitySmalltalk inspect*).

Básicamente los mensajes de esta clase permiten:

En cuanto a las fuerzas:
- Obtener una fuerza con un determinado nombre.
- Obtener la lista de nombres de todas las fuerzas existentes.
- Obtener el organizador de una determinada fuerza. (veremos esto con más detalle cuando analicemos la clase “Force”), para saber cuáles son los determinantes definidos para cada clase.
- Averiguar si existe una fuerza con un nombre determinado.
- Agregar una fuerza con un nombre dado a la colección de fuerzas existentes.
- Eliminar una determinada fuerza de la colección de fuerzas existentes.

En cuanto a los bloques de lógica de fuerzas:
- Almacenar un bloque de lógica de fuerzas para una clase dada.
- Almacenar un bloque de lógica de fuerzas para una clase y un mensaje dados.
- Eliminar todos los bloques de lógica de fuerza de una clase dada.
- Eliminar el bloque de lógica de fuerza de una clase y un mensaje dados.
- Obtener la fuerza y el determinante que predominan en un contexto de ejecución dado.
- Obtener el bloque de lógica de fuerzas de una clase dada.
- Obtener el bloque de lógica de fuerzas de una clase y un mensaje dados.

En realidad la mayoría de estos mensajes se resuelven redireccionando los mensajes a los objetos de las variables de instancia correspondientes.

Las Fuerzas (Clase “Force”)

La clase “Force” representa las fuerzas que determinan la existencia de objetos subjetivos. Cada fuerza existente es una instancia de esta clase.

Esta clase tiene definida dos variables de instancia:
- name: es el nombre de la fuerza (por ejemplo: “Estado”, “Emisor”, etc.).
- forceOrganizer: es una instancia de la clase ClassOrganizer y es la encargada de almacenar los determinantes de la fuerza para todas las clases que definen determinantes a la misma (por ejemplo: la fuerza “Estado” sabe que los determinantes de la clase “Stack” son “Vacío”/”Lleno”, los de la clase “LimitedStack” son “Vacío”/”Lleno”/”Intermedio”, los de la clase “CuentaCorriente” son “EnRojo”/”ConFondo”, y así sucesivamente).

Básicamente los mensajes de esta clase permiten:

- Setear y Obtener el nombre de la fuerza.
- Setear y Obtener el organizador que contiene los determinantes de la fuerza para todas las clases/métodos que tengan determinantes definidos.

Las fuerzas creadas en el ambiente son almacenadas en la instancia “SubjectivitySmalltalk”, como ya se explicó previamente.

El Mecanismo de Despacho de Mensajes

Hasta ahora sabemos que para resolver la Subjetividad planteada en este trabajo necesitamos poder guardar de alguna forma los diferentes métodos asociados a un mismo mensaje, que serán ejecutados bajo las diferentes fuerzas y determinantes influyentes.

¿ Cómo se almacenan y despachan los métodos en la implementación estándar de Smalltalk?

Todos los objetos son instancias de una clase; las clases mismas deben estar representadas por instancias de una clase. La clase *Class* es donde se define cuál es el comportamiento y la estructura de datos que tiene cualquier clase de Smalltalk. Parte de la estructura de datos está compuesta por el diccionario de métodos (instancia de la clase *MethodDictionary)* cuyas claves son los nombres de los mensajes (*selectores),* y el valor asociado a cada clave es un método compilado. El objeto que contiene a un método compilado es una instancia de la clase *CompiledMethod.*

Veamos gráficamente esta situación:

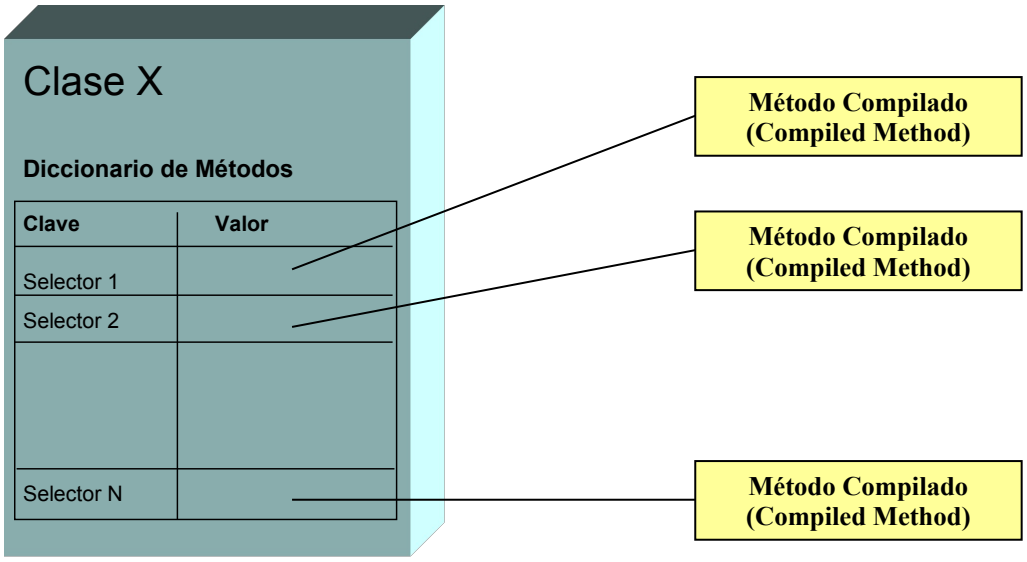


Figura 15: Almacenamiento de Métodos Compilados en un Ambiente Smalltalk No Subjetivo

Esta estructura encaja perfectamente cuando a cada mensaje se le asocia un solo método. Pero esto ya no es tan así, si necesitamos asociar varios métodos a un mismo mensaje, como es el caso de los mensajes subjetivos.

Ahora debemos tener en cuenta que un método subjetivo, por tratarse de una de las posibles implementaciones asociadas a un mensaje subjetivo, no puede almacenarse directamente en el diccionario de métodos de la clase, pues este diccionario permite una sola entrada por cada mensaje (selector)[6] , y nosotros necesitamos almacenar varias implementaciones para un mismo mensaje. Además se agrega otro nivel de complejidad: al momento de ejecución se necesita poder determinar de alguna forma cuál de todas las implementaciones ejecutar.

Esto lo podemos resolver de la siguiente manera: en el caso de los métodos subjetivos lo que debe guardarse en el diccionario de métodos de la clase es un objeto (instancia de la clase "SubjectivityCompiledMethod"), cuyo método compilado corresponderá a un código que permita determinar en tiempo de ejecución, de acuerdo con la lógica de fuerzas, cuál es la implementación adecuada para ejecutar. Por eso, el "SubjectivityCompiledMethod" deberá contener este código de determinación, y además todos los métodos definidos para el mensaje subjetivo, cada uno de los cuales estará asociado a la fuerza y el determinante correspondiente. (Veremos esto más en detalle cuando analicemos la clase "SubjectivityCompiledMethod"). El "SubjectivityCompiledMethod" se encontrará entonces en el diccionario de métodos de la clase subjetiva, y su clave será, como en el caso del "CompiledMethod", el nombre del mensaje (selector).

Resumiendo, para poder definir métodos subjetivos, creamos la clase *SubjectivityCompiledMethod*. En caso que un método se implemente en forma subjetiva, en el diccionario de métodos de la clase subjetiva correspondiente se almacenará un "SubjectivityCompiledMethod" en lugar de un "CompiledMethod".

Veamos entonces gráficamente la nueva situación:

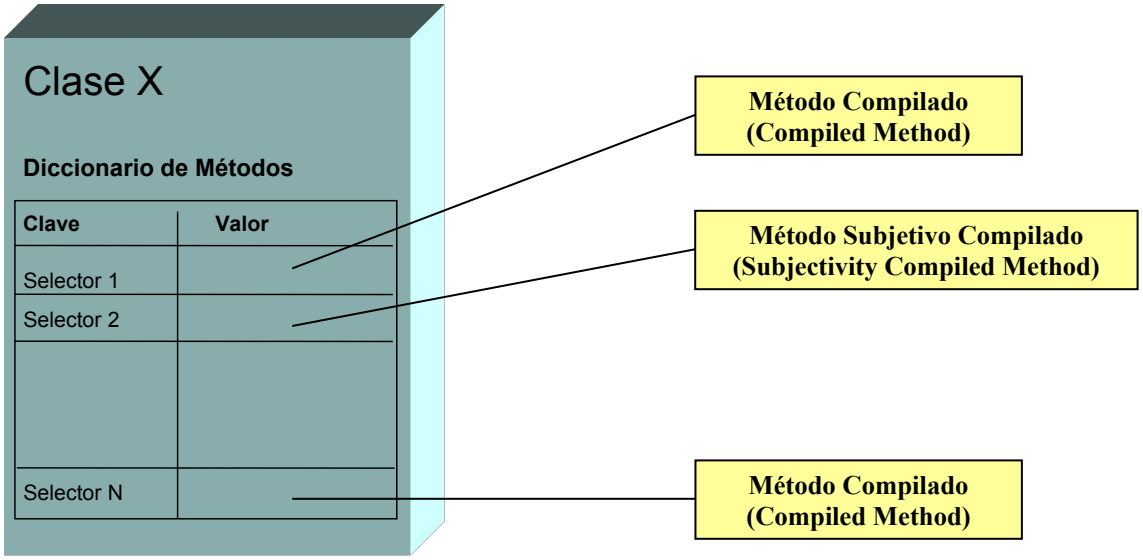


Figura 16: Almacenamiento de Métodos Compilados en un Ambiente Smalltalk Subjetivo

[6] Notar que esto sucede porque justamente el selector (nombre del mensaje) es la clave del diccionario de métodos.

El SubjectivityCompiledMethod

La clase “SubjectivityCompiledMethod” es subclase de “SubjectiveMethod”, quien a su vez es subclase de “CompiledMethod”. Extiende el comportamiento del “CompiledMethod” haciendo que en el momento de recibir el mensaje se pueda despachar la implementación correcta. Es decir que cuando un objeto recibe un mensaje, se realiza la búsqueda habitual del método[7] en el diccionario de métodos de la clase. Al encontrarlo, se ejecuta el método. En caso que este método sea una instancia de SubjectivityCompiledMethod, existe la posibilidad de ejecutar diferentes CompiledMethod’s, según la fuerza y determinante vigentes en el momento de ejecución.

Antes de continuar con el SubjectivityCompiledMethod y a los fines de entender como funciona el SubjectivityCompiledMethod, explicaremos brevemente qué es un CompiledMethod y qué es un Context.

En Smalltalk, el código fuente de los métodos escritos por el programador está representado por instancias de la clase Text. Este código fuente es traducido por el *Compilador* en secuencias de instrucciones para un *Intérprete*, denominadas *bytecodes* por ser números de ocho bits.

El programador no interactúa directamente con el *Compilador* sino que cuando un nuevo método es agregado a una clase, esta clase le pide al *Compilador* una instancia de la clase CompiledMethod conteniendo la traducción a bytecodes del método fuente agregado.

La ejecución de un método, consiste en la ejecución de los bytecodes asociados al CompiledMethod por parte del *Intérprete.* Este *Intérprete* necesitará cierta información para poder controlar esta ejecución (por ejemplo, el receptor del mensaje, los argumentos del mismo, cuáles son los bytecodes ejecutados, etc.).

Durante la ejecución de un método, al enviar un nuevo mensaje, el estado del *Intérprete* cambiará a fin de ejecutar otro CompiledMethod, en respuesta a este nuevo mensaje. Estos estados del *Intérprete* son guardados en objetos denominados contextos (objetos de la clase *Context*).

En un momento dado, pueden haber varios contextos en el sistema. El contexto que representa el estado actual del *Intérprete* es denominado “contexto activo”. Para referirse a un contexto activo dentro del código fuente, se puede utilizar la pseudovariable *thisContext.* Cuando un bytecode determina que el CompiledMethod del contexto activo requiere de la ejecución de un nuevo CompiledMethod, el contexto activo se suspende y se crea un nuevo contexto que pasa a ser el activo.

Realizada esta explicación, volvamos a la implementación de la clase SubjectivityCompiledMethod.

Esta clase tiene la siguiente estructura: un código (bytecodes) y un diccionario.

En el diccionario hay una entrada para cada una de las fuerzas en las cuales hay algún método definido. El valor de cada componente del diccionario será un segundo diccionario: en este caso, hay una entrada para todos de los determinantes de la fuerza.

El valor asociado a cada determinante dependerá si el programador definió o no, un método para ese determinante.
En caso de haberlo definido, el valor asociado al determinante será el método compilado (CompiledMethod) que se ejecutará en caso de que la fuerza y el determinante resulten vigentes.
En el otro caso, cuando el programador no define el método, el valor asociado será el método nulo (NullCompiledMethod). Este último caso permite que al no definir una implementación para una fuerza y determinante en particular, se ejecute el método definido en la superclase.

[7] La búsqueda del método en el diccionario de métodos de la clase se conoce comúnmente como “Method Look-Up”.

El código (bytecodes) asociado a un SubjectivityCompiledMethod permite que en tiempo de ejecución se le solicite a la lógica de fuerzas determinar cuál es la fuerza predominante en ese instante, y en base a ello despachar la ejecución del método correspondiente.

A continuación podemos observar gráficamente la estructura que posee un SubjectivityCompiledMethod.

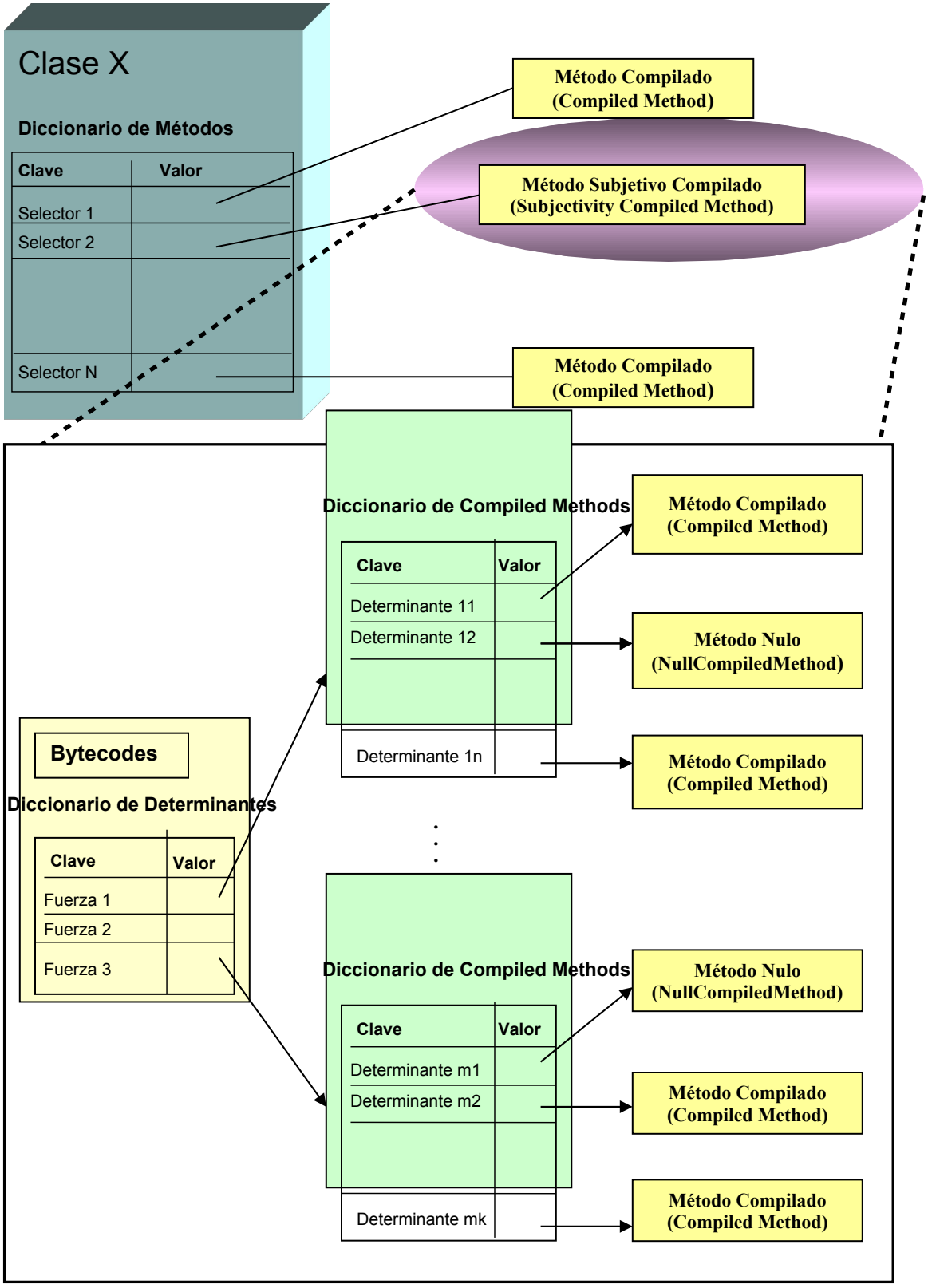


Figura 17 : Estructura de un SubjectivityCompiledMethod

El browser subjetivo es el encargado de generar el SubjectivityCompiledMethod al momento de almacenar un método subjetivo.

Todos los SubjectivityCompiledMethod tienen un código fuente similar. La única diferencia entre dos SubjectivityCompiledMethod's distintos puede llegar a ser el nombre del mensaje subjetivo (selector) que se desea ejecutar. Es decir, cuando un SubjectivityCompiledMethod se crea, se generan los bytecodes correspondientes a un método que tiene como nombre el mismo nombre del mensaje subjetivo y representa la traducción del código fuente correspondiente a la figura que se muestra a continuación.

```
        | aCompiledMethod    methodArguments    anExecutionContext |

(1)     methodArguments := Array new: (thisContext method numArgs).
        1 to: (thisContext method numArgs) do: [:i | methodArguments at: i put: (thisContext localAt: i)].

(2)     anExecutionContext:= ExecutionContext new.
        anExecutionContext subjectClass: (thisContext method mclass).
        anExecutionContext subject: self.
        anExecutionContext sender: ( (thisContext sender) receiver).
        anExecutionContext arguments: methodArguments.
        anExecutionContext selector: (thisContext selector).

(3)     aCompiledMethod  :=  (thisContext method) subjectiveMethodFor: anExecutionContext.
(4)     ^aCompiledMethod valueWithReceiver: (thisContext receiver) arguments: methodArguments.
```

Figura 18 : Código que ejecutan todos los SubjectivityCompiledMethod

Veamos la lógica de este código, analizando cada uno de los ítems remarcados.

(1) Se arma un array que contiene todos los argumentos (parámetros) existentes en la invocación del mensaje.

(2) Se crea el objeto "anExecutionContext" (instancias de la clase "ExecutionContext"), y se guardan los valores del contexto de ejecución activo (*thisContext)* a los fines que puedan ser utilizados para obtener el CompiledMethod en el ítem 3; estos valores son:
- Nombre de la clase a la que pertenece el objeto receptor del mensaje (subjectClass).
- Objeto receptor del mensaje (subject)
- Objeto emisor del mensaje (sender)
- Argumentos (parámetros) del mensaje (arguments).
- Selector asociado al mensaje subjetivo (selector)

En la sección del Contexto de ejecución se explican los motivos por los cuales es crear el objeto "anExecutionContext".

(3) Se obtiene el método compilado que corresponde ejecutar según la fuerza y el determinante que prevalecen en el momento de invocación del mensaje. Para determinar esto, se recurre a la lógica de fuerzas, que será evaluada de acuerdo con los valores del contexto de ejecución activo ("anExecutionContext").

(4) Se ejecuta del método compilado obtenido en el ítem (3); como puede observarse se pasan los parámetros existentes en la invocación del mensaje.

Básicamente los mensajes definidos en la clase "SubjectivityCompiledMethod" permiten:

- Agregar un método compilado bajo una determinada fuerza y determinante del SubjectivityCompiledMethod.
- Agregar el método compilado nulo bajo todos los determinantes definidos.
- Obtener el método compilado bajo una determinada fuerza y determinante del SubjectivityCompiledMethod.
- Cambiar el nombre de una fuerza dentro del SubjectivityCompiledMethod.
- Obtener la fuerza y el determinante bajo el cual se encuentra definido un método dado (para debug).
- Averiguar si el SubjectivityCompiledMethod tiene métodos asociados.
- Averiguar si el SubjectivityCompiledMethod incluye un método determinado.
- Eliminar un determinante asociado a una fuerza dentro del SubjectivityCompiledMethod.
- Obtener el método compilado que corresponde ejecutar según la fuerza y el determinante que prevalecen en el momento de invocación del mensaje. Para determinar esto, se recurre a la lógica de fuerzas, que será evaluada de acuerdo con los valores del contexto de ejecución ("anExecutionContext").

El Método Nulo (Clase "NullCompiladoMethod")

La clase "NullCompiledMethod" también es subclase de "SubjectiveMethod". Permite definir un método que simplemente hace un ^*super* de sí mismo. Es decir: es un método que al ser ejecutado hace que se ejecute el método definido en alguna superclase con su mismo nombre de mensaje (selector).

Como veremos posteriormente en la sección dedicada al browser subjetivo, un SubjectivityCompiledMethod se crea cuando el programador define el código fuente para una fuerza y determinante. En el momento de su creación, el SubjectivityCompiledMethod inicializa todas las entradas asociadas al diccionario de determinantes con instancias de NullCompiledMethod, y la entrada asociada a la fuerza y determinante que seleccionó el programador, con el CompiledMethod correspondiente al código fuente definido.

Este forma de trabajar provoca que, para los determinates que no tengan implementación definida, se recurra -mediante el mecanismo de herencia habitual- a la implementación del mensaje definida por la superclase.

El NullCompiledMethod contiene solamente un código (bytecodes -de manera similar al SubjectivityCompiledMethod-) correspondientes a un método que tiene como nombre el mismo nombre del mensaje subjetivo, y representa la traducción del siguiente código fuente:

^super *<<aFullSelector>>*

donde <<*aFullSelector*>> representa el nombre completo del método (incluyendo los parámetros).

Retomando el ejemplo del LimitedStack. Asumamos que aún no fue definido el método para "push". En el momento que se crea el método "push" para el determinante "Lleno", se inicializan autómáticamente métodos para los determinantes "Intermedio" y "Vacío" con NullCompiledMethod's. La siguiente tabla, muestra que tipo de método estará a asociado a cada determinante y el código fuente de ese método.

MENSAJE \ ESTADO	***VACIO***	***INTERMEDIO***	***LLENO***
push: anElement	NullCompiledMethod para el código: ^*super push:anElement*	NullCompiledMethod para el código: ^*super push:anElement*	CompiledMethod para el código: ^*nil.*

El "SubjectiveMethod"

La clase "SubjectiveMethod" es una clase abstracta, subclase de "CompiledMethod". Esta clase se diseñó con el objetivo de poder compartir el comportamiento común que existe entre el SubjectivityCompiledMethod y el NullCompiledMethod.

Este comportamiento común está relacionado principalmente con el hecho de tener unos "bytecodes" que dependen de la clase concreta en la que se generen. Por este motivo, los mensajes de esta clase permiten:

- Crear un método subjetivo a partir de una clase y el nombre del mensaje con los parámetros.
- Retornar el código para derivar la ejecución al método compilado asociado a la fuerza predominante.
- Guardar los bytecodes a partir de un método compilado.
- Inicializarse a partir de una clase, el nombre del mensaje con los parámetros y un método compilado.
- Cambiar dentro del diccionario de métodos de una clase al método compilado existente por él.

El Contexto de Ejecución (Clase "ExecutionContext")

La clase "ExecutionContext" representa los contextos de ejecución necesarios para poder llevar a cabo la subjetividad. Ante la ejecución de un mensaje subjetivo se crea un objeto de esta clase; el mismo encapsula la información necesaria del contexto activo para determinar - en tiempo de ejecución del mensaje - cuál es la implementación que debe ejecutarse de acuerdo con la fuerza que prevalece en dicho momento.

Esta clase tiene definida las siguientes variables de instancia:

- subjectClass: Nombre de la clase a la que pertenece el objeto receptor del mensaje
- subject: Objeto receptor del mensaje
- sender: Objeto emisor del mensaje
- arguments: Argumentos (parámetros) del mensaje
- selector: Selector asociado al mensaje subjetivo

Básicamente los mensajes de esta clase permiten:

- Setear y Obtener la clase a la que pertenece el objeto receptor del mensaje.
- Setear y Obtener el objeto receptor del mensaje subjetivo.
- Setear y Obtener el objeto emisor del mensaje subjetivo.
- Setear y Obtener los argumentos involucrados en el mensaje subjetivo.
- Setear y Obtener el selector asociado al mensaje subjetivo.

Esta información es utilizada fundamentalmente en la lógica de fuerza, como veremos más adelante, y permitirá tomar la decisión de qué fuerza y determinante prevalece en el momento de invocación del mensaje subjetivo; esto definirá qué implementación ejecutar.

El encapsulamiento de la información necesaria para el contexto de ejecución brindado por esta clase permite una fácil extensión del ambiente ante la aparición de nuevas fuerzas con características distintas a las actuales, permitiendo agregarle el comportamiento necesario para contemplar esas nuevas características.

En el Smalltalk estándar existen las clases MethodContext y BlockContext, subclases de Context, que representan el contexto para ejecutar un método como resultado del envío de un mensaje o para ejecutar un bloque, respectivamente. Este contexto contiene la información del emisor del mensaje, del receptor, de los argumentos, etc. en el momento en el que un método o un bloque es ejecutado. Cada ejecución de un método es acompañada por un MethodContext. Cada evaluación de un bloque es acompañada por un BlockContext. Cuando se invoca a la lógica de fuerzas, el contexto de ejecución no debe variar, se debe preservar el emisor del mensaje, los argumentos, etc.

El SubjectivityCompiledMethod tiene asociado al ejecutarse un MethodContext y el bloque de lógica de fuerzas un BlockContext. Cuando el SubjectivityCompiledMethod solicita la fuerza y determinante que predominan al momento de la ejecución, termina generando la evaluación del bloque de lógica de fuerzas, por parte de la misma lógica de Fuerzas (clase ForceLogicDefinition). Es decir que dentro del BlockContext del bloque de lógica de Fuerzas, el sender es un objeto de la clase ForceLogicDefinition. Como se puede observar, dentro del bloque de lógica de Fuerzas nos interesa el contexto activo al momento de recibir el mensaje, a los efectos de poder tomar la decisión adecuada. Es decir, que nos interesa la información del MethodContext correspondiente al SubjectivityCompiledMetodo que se está ejecutando. Esta información necesaria es la se encuentra “congelada” en el ExecutionContext.

La Lógica de Fuerzas (el bloque de la lógica y las Clases “ForceLogicDefinition” e “InfluenceForce”)

Ya sabemos que la lógica de fuerzas es “*el algoritmo que determina qué fuerza y determinante prevalecen sobre un objeto subjetivo en un instante dado*”, y que ésta puede definirse a nivel de clase o de método.

Dado que la lógica de fuerzas de cada clase/método es definida por el programador, decidimos que la especificación de la misma se realice en un bloque de código con parámetros predeterminados. El formato del bloque está dado por una plantilla que se muestra al programador cuando no existe lógica de fuerzas definida. A continuación podemos ver el formato que debe poseer la lógica de fuerza:

```
[:anExecutionContext |
        | anInfluenceForce |
        anInfluenceForce := InfluenceForce new.
         " >> Put your logic definition here <<
              You can define this logic using the values specified in anExecutionContext.
              For more information, see the ExecutionContext class comment.
         "
         anInfluenceForce force: “aForce”.
         anInfluenceForce determinant: “aDeterminant”.
         anInfluenceForce.
].
```

Como podemos observar, el bloque recibe como parámetro el objeto “anExecutionContext” (instancia de la clase “ExecutionContext”), conteniendo toda la información necesaria para que el programador asocie a una determinada situación la fuerza y el determinante que predominan sobre el objeto en ese instante de ejecución.

El objeto “anExecutionContext” contiene la información correspondiente al contexto de ejecución, y permite obtener, entre otras cosas, el receptor y el emisor del mensaje, así como también los parámetros de invocación del mismo. El programador puede indagar estos objetos para determinar qué fuerza y determinante prevalecen en determinado momento.

La evaluación del bloque debe retornar como resultado el objeto “anInfluenceForce” (instancia de “InfluenceForce”), representando a la fuerza que influye sobre el objeto receptor del mensaje. El objeto “anInfluenceForce” tiene la información de la fuerza y el determinante bajo los cuales debe buscarse el método compilado que debe ejecutarse.

Retomemos nuestro ejemplo del "Stack", y veamos cómo se definió la lógica de fuerzas para esta clase.

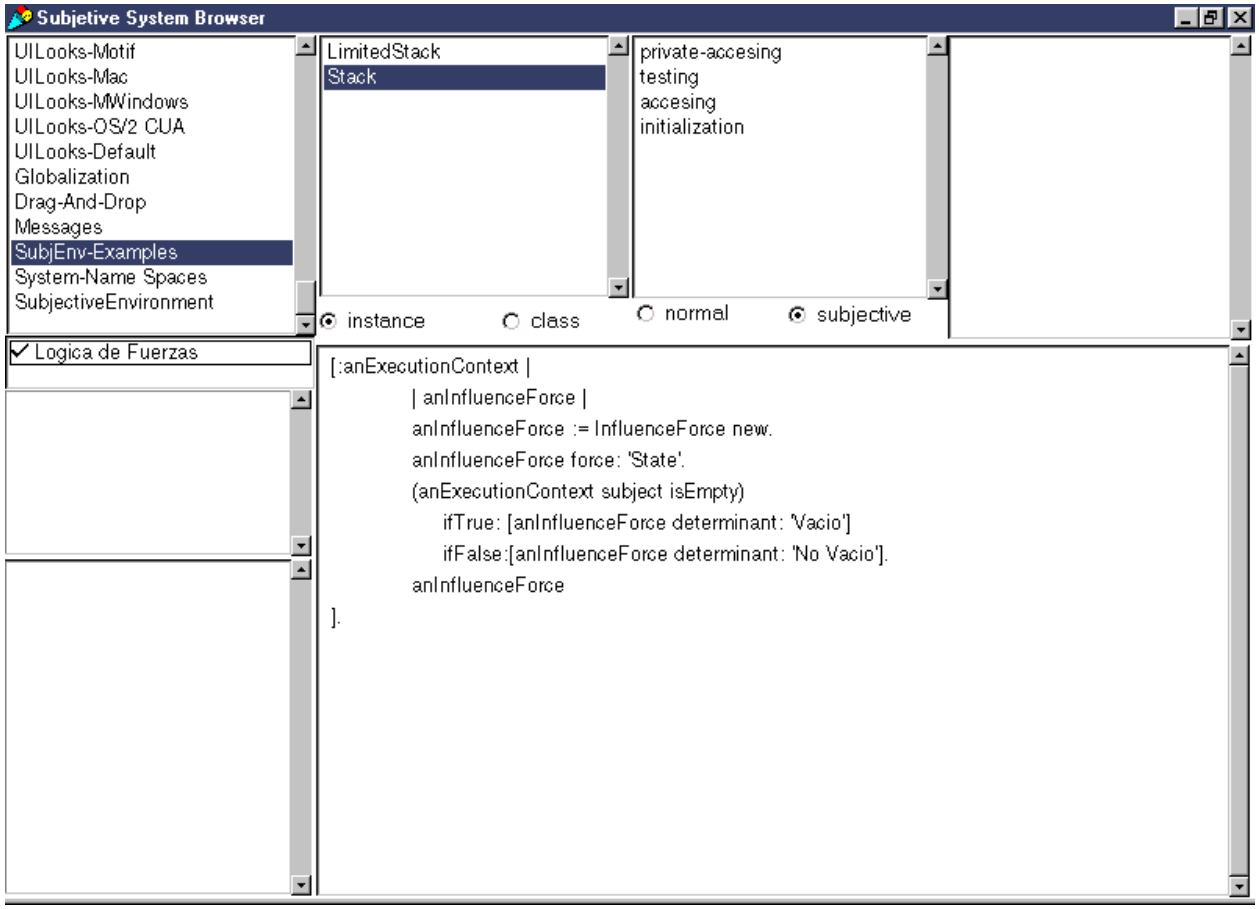


Figura 19: Definición de la Lógica de Fuerzas para la clase Stack.

Como podemos observar, la fuerza que predomina en todo momento es la fuerza "State" ("Estado"). Pero en cambio, el determinante varía de acuerdo con la cantidad de elementos del stack. Por eso el programador indaga al objeto receptor del mensaje ("anExecutionContext subject") sobre su cantidad de elementos ("isEmpty"). Si el stack no contiene elementos, entonces el determinante de la fuerza "State" es "Vacio"; de otro modo es "No Vacio".

También podemos notar que esta lógica de fuerzas se encuentra definida a nivel de clase, ya que solamente está seleccionada la clase "Stack" (y no se encuentra seleccionado ningún método en particular).

Veamos ahora el caso del "LimitedStack" y la lógica de fuerzas definida para esa clase.

Como podemos observar en la figura correspondiente, aquí también la fuerza que predomina en todo momento es la fuerza "State" ("Estado"), y el determinante varía de acuerdo con la cantidad de

elementos del stack, solamente que ahora se tienen en cuenta los estados propios del LimitedStack: "Vacio", "Intermedio" y "Lleno".

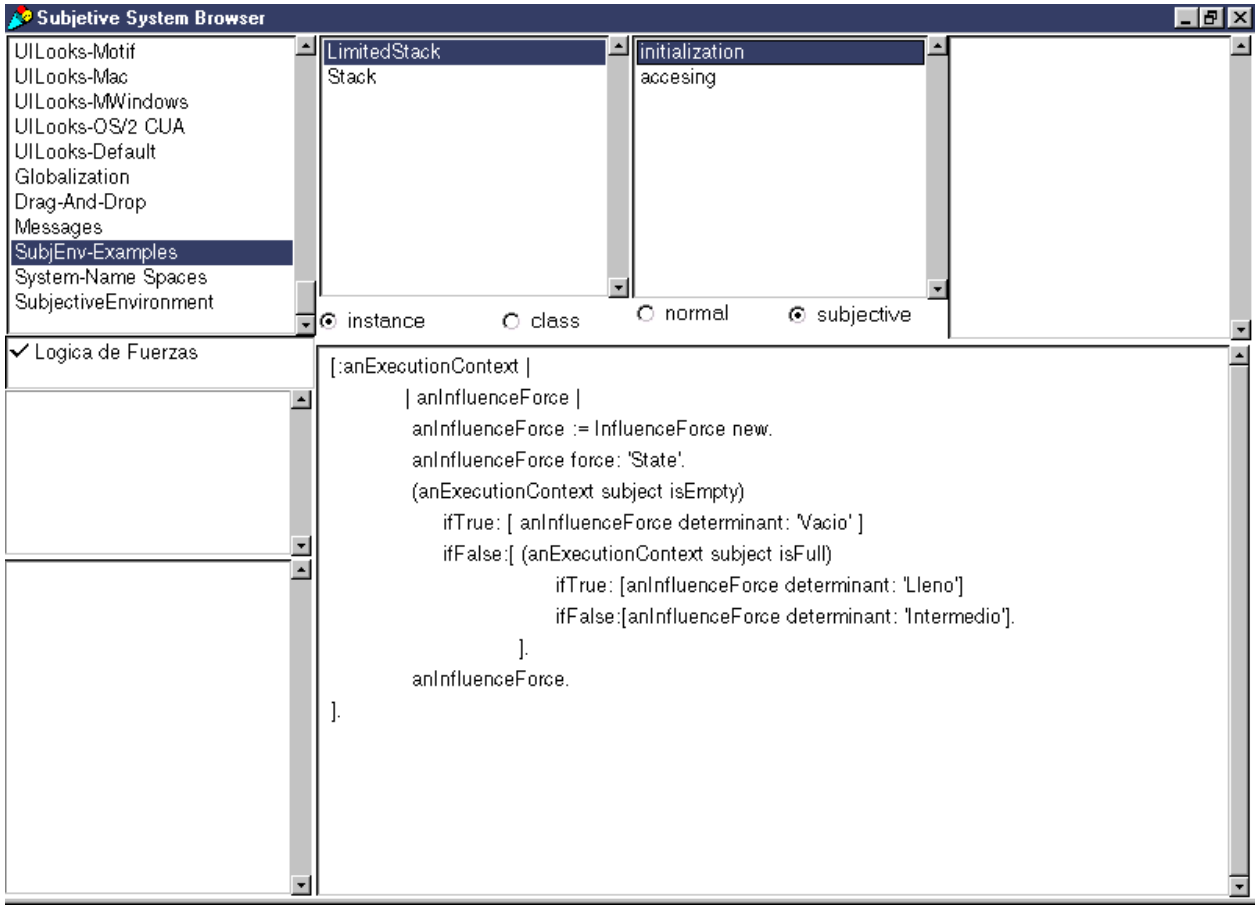


Figura 20: Definición de la Lógica de Fuerzas para la clase LimitedStack.

Si bien ya lo explicamos en el método de despacho de mensajes y en el "SubjectivityCompiledMethod", debemos recordar que este bloque de lógica de fuerzas se evalúa sobre el objeto receptor del mensaje ante cada invocación de un mensaje subjetivo. Esto debe ser así, porque la situación del stack puede variar entre cada invocación de mensaje.

Esta metodología sería equivalente a una transición de estados, solamente que uno no sabe de antemano cuál es el estado en el que está, sino que debe calcularlo. Para el caso de la fuerza de Estado podría almacenarse el estado resultante de la ejecución del mensaje con el objetivo de que el objeto receptor quede preparado para el próximo mensaje a recibir. Esto funciona para la fuerza de Estado, pero no es válido, por ejemplo, para la fuerza del Emisor, ya que no se puede predecir quién será el emisor del próximo mensaje recibido.

Por este motivo no realizamos este estilo de implementación, y debe evaluarse nuevamente la lógica de fuerzas ante cada recepción de mensaje subjetivo.

¿ Cómo se almacenan los bloques de lógica de fuerzas?

Los bloques de lógica de fuerzas se almacenan en un objeto que es una instancia de la clase "ForceLogicDefinition".

Esta clase posee una variable de instancia denominada "logicBlocks" que en realidad es un diccionario.
Como cada clase subjetiva posee sus propias lógicas de fuerzas (a nivel de clase/método), la clave del diccionario es el nombre de cada clase subjetiva que tenga definido por lo menos un bloque de lógica de fuerzas.
El valor asociado a cada una de estas claves es a su vez otro diccionario, donde efectivamente se almacenarán los bloques de texto. Este segundo diccionario tiene como clave o bien el nombre de la clase (si el bloque es a nivel de clase), o bien el nombre del selector (si el bloque es a nivel de método).

Los bloques en sí se guardan en el segundo diccionario, asociados a la clave correspondiente de acuerdo con el alcance del bloque de lógica de fuerzas.

Veamos gráficamente un esquema de esta estructura:

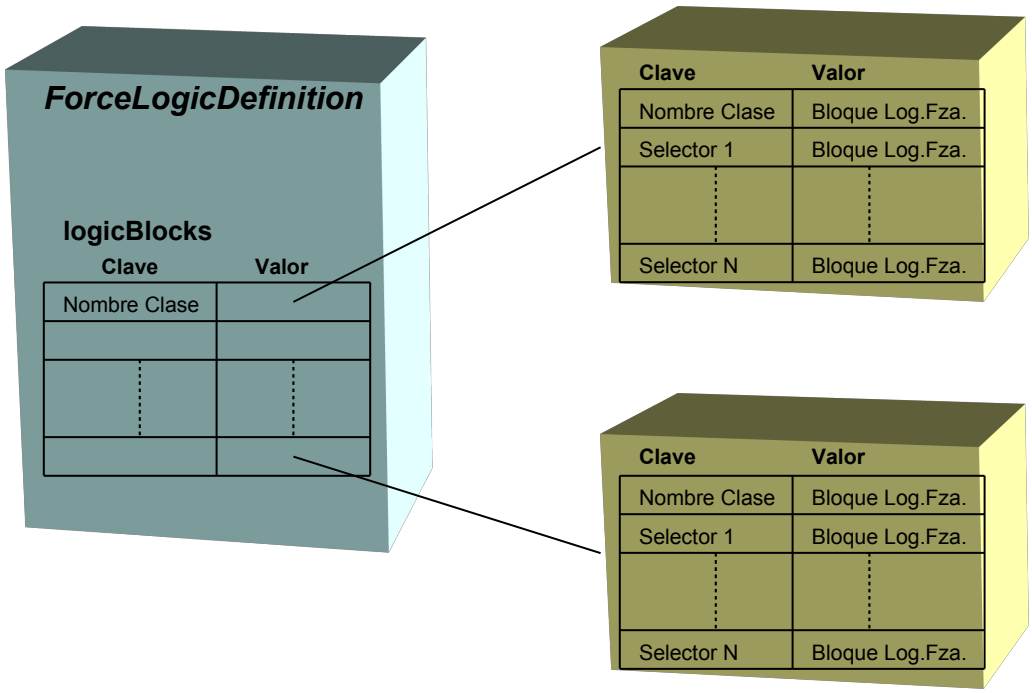


Figura 21: Almacenamiento de Bloques de Lógica de Fuerzas

Básicamente los mensajes definidos en la clase “ForceLogicDefinition” permiten:

- Agregar un bloque de lógica de fuerzas para una determinada clase y selector (en caso de selector nulo, se asume que el alcance del bloque es a nivel de clase).

- Obtener el bloque de lógica de fuerzas definido para una determinada clase y selector (en caso de selector nulo, se asume que el alcance del bloque es a nivel de clase).

- Obtener la fuerza y el determinante que predominan en un momento dado para un objeto subjetivo de acuerdo con un contexto de ejecución determinado.

  Para esto se definió el mensaje denominado “getInfluenceForce:anExecutionContext”. Como el “anExecutionContext” tiene la clase a la que pertenece el receptor del mensaje y el selector correspondiente al mensaje invocado, los utiliza como claves en los diccionarios y de esta manera obtiene el bloque de lógica de fuerzas que corresponde. Finalmente evalúa el bloque pasándole como parámetro el “anExecutionContext”; esta evaluación retorna un objeto de la clase “InfluenceForce” conteniendo la fuerza y el determinante que prevalecen para el receptor del mensaje en ese contexto de ejecución.

- Obtener el selector asociado a un bloque de lógica de fuerzas en una clase dada.

- Eliminar todos los bloques de lógica de fuerzas definidos para una determinada clase.

- Eliminar el bloque de lógica de fuerzas definido para una clase y un selector determinados (si el selector es nulo, se asume que se desea eliminar el bloque de lógica de fuerza a nivel de clase).

Este objeto “forceLogicDefinition” encargado de almacenar los bloques de lógica de fuerzas, es parte del objeto global “SubjectivitySmalltalk” que explicáramos con anterioridad.

Ejecución de un Mensaje Subjetivo

A continuación mostramos la secuencia de ejecución de un método subjetivo, en la cual se encuentran involucrados varios de los componentes del motor de subjetividad:

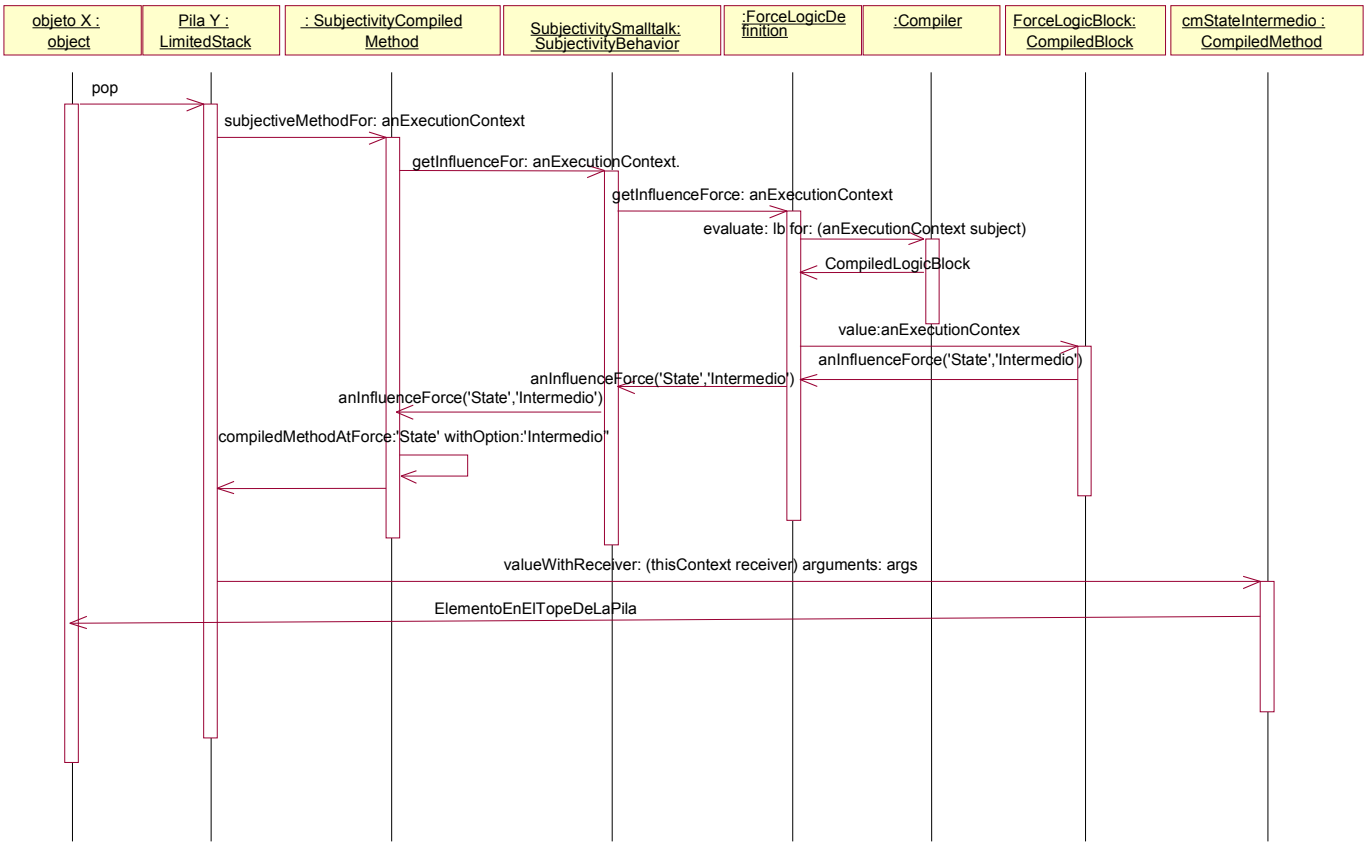


Figura 22 : Secuencia de ejecución de un mensaje Subjetivo

Herramientas de desarrollo del ambiente subjetivo

El Administrador de Fuerzas (Clase “ForceAdministrator”)

Una de las primeras herramientas necesarias en el ambiente, es una herramienta que permita fácilmente agregar, eliminar y cambiar las fuerzas definidas dentro del ambiente.
Para esto se creó la herramienta denominada “Forces Administrator”. La misma puede accederse de dos lugares diferentes:

a. Desde el “Launcher”, opción de menú “Browse”\”Admin Forces”.
b. Desde el menú desplegado con el botón derecho del mouse, que aparece en el “Browser del ambiente subjetivo”, más precisamente en el pane correspondiente a las fuerzas existentes en el sistema.

La herramienta consiste de una simple ventana que despliega los nombres de las fuerzas existentes, y permite (previo ingreso / selección de texto) el agregado, la modificación y la eliminación de fuerzas. Cabe aclarar que las opciones de eliminación y modificación NO implican la modificación de las fuerzas dentro de los bloques de lógica de fuerza, porque dicha consideración implican un análisis semántico del texto escrito por el programador en dicho bloque.

Veamos cómo se ve esta herramienta en forma gráfica:

a. Desplegado desde el “Launcher”:

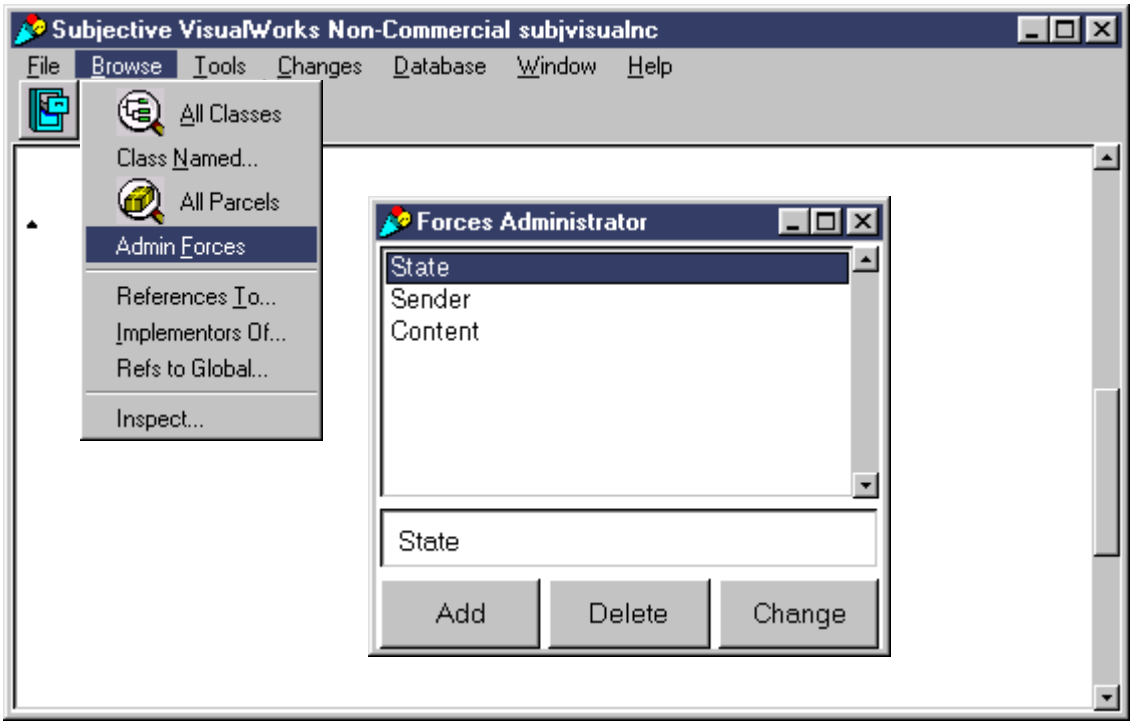


Figura 23 : Apertura del Administrador de Fuerzas desde el VisualLauncher Subjetivo

b. Desplegado desde el “Browser del ambiente subjetivo”:

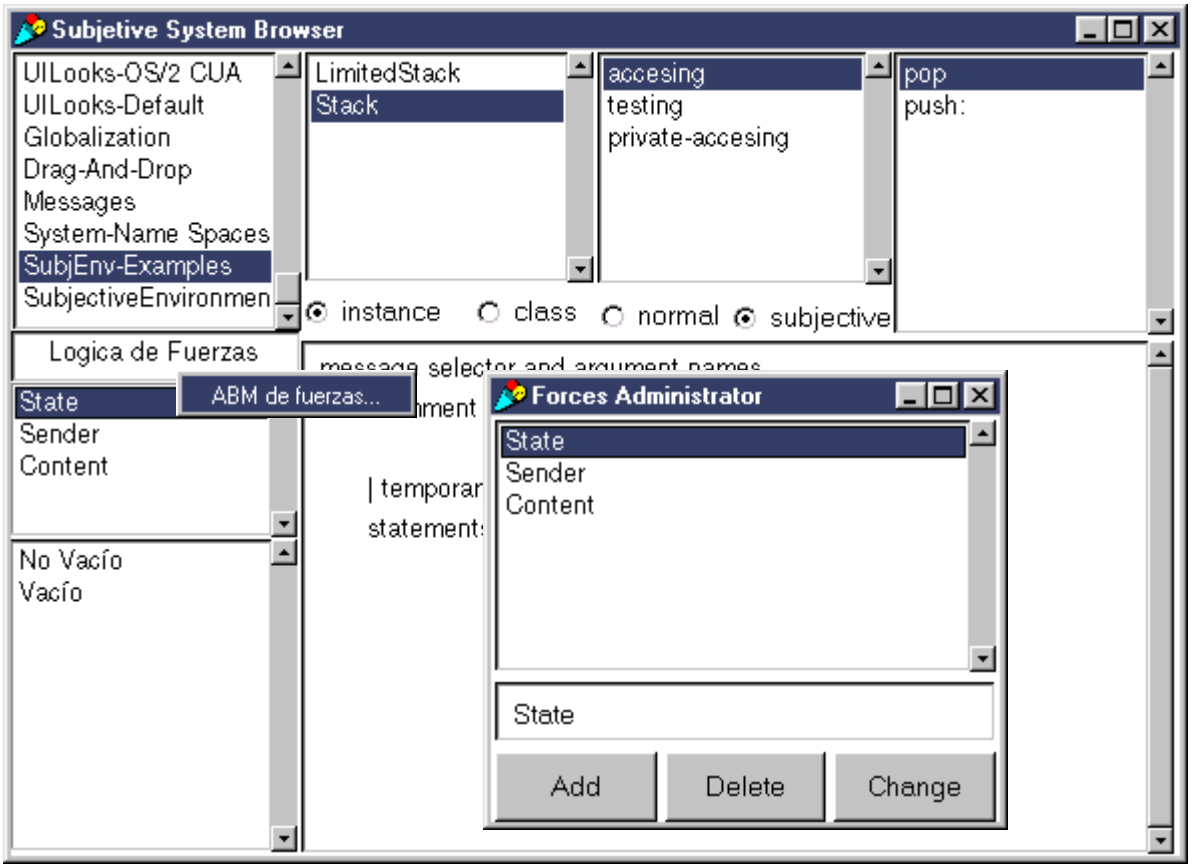


Figura 24: Apertura del Administrador de Fuerzas desde el Browser Subjetivo

El Browser del ambiente Subjetivo (Clases “Browser” y “SubjectivityBrowser”)

Aquí vale la pena realizar la siguiente aclaración: el nuevo browser reemplaza al browser tradicional, debido a que solamente de esta forma se podrá mantener la consistencia del ambiente subjetivo. Esto implica modificar el tipo de browser que abre el Launcher del Smalltalk. De no ser así, el motor de subjetividad implementado no sería transparente para el programador, pudiendo inclusive generar inconsistencias en el ambiente. Por ejemplo: si realiza modificaciones a métodos subjetivos utilizando el browser tradicional, perdería las implementaciones subjetivas del mensaje en cuestión.

Una de las herramientas con las que claramente se necesita contar es con un browser que le brinde al programador un acceso claro y amigable a la información del nuevo ambiente. Como explicáramos en párrafos anteriores, no todas las clases tienen la obligación de ser subjetivas; de hecho esto no sucede en el mundo real, puesto que algunos objetos responden siempre de la misma manera a un mismo mensaje (recordar el ejemplo de los números). Esto implica que este nuevo browser, debe mantener la funcionalidad actual (para realizar todas las operaciones que se realizan hoy en día sin manejo de subjetividad), y además agregar nueva funcionalidad que facilite la definición y manejo en general del ambiente subjetivo.

La clase “SubjectivityBrowser”, subclase de “Browser”, fue creada para tal fin. La misma permite básicamente la edición y compilación de los métodos “normales” (no subjetivos) y agrega la edición y compilación de los métodos “subjetivos”, como así también de la lógica de fuerzas.

Vayamos viendo gráficamente las posibilidades que nos brinda este nuevo browser.

Manejo de la misma funcionalidad que se cuenta en la actualidad para trabajar con clases “normales” (no subjetivas).

Para esto alcanza con seleccionar el botón “normal” ubicado debajo de la ventana de “categorías”. Como se puede observar en la siguiente imagen, el browser es prácticamente análogo al que uno está acostumbrado a manejar en la actualidad, y de hecho permite exactamente las mismas operaciones.
En el ejemplo, se muestra el método correspondiente al mensaje “at: put:” de la clase “OrderedCollection”.

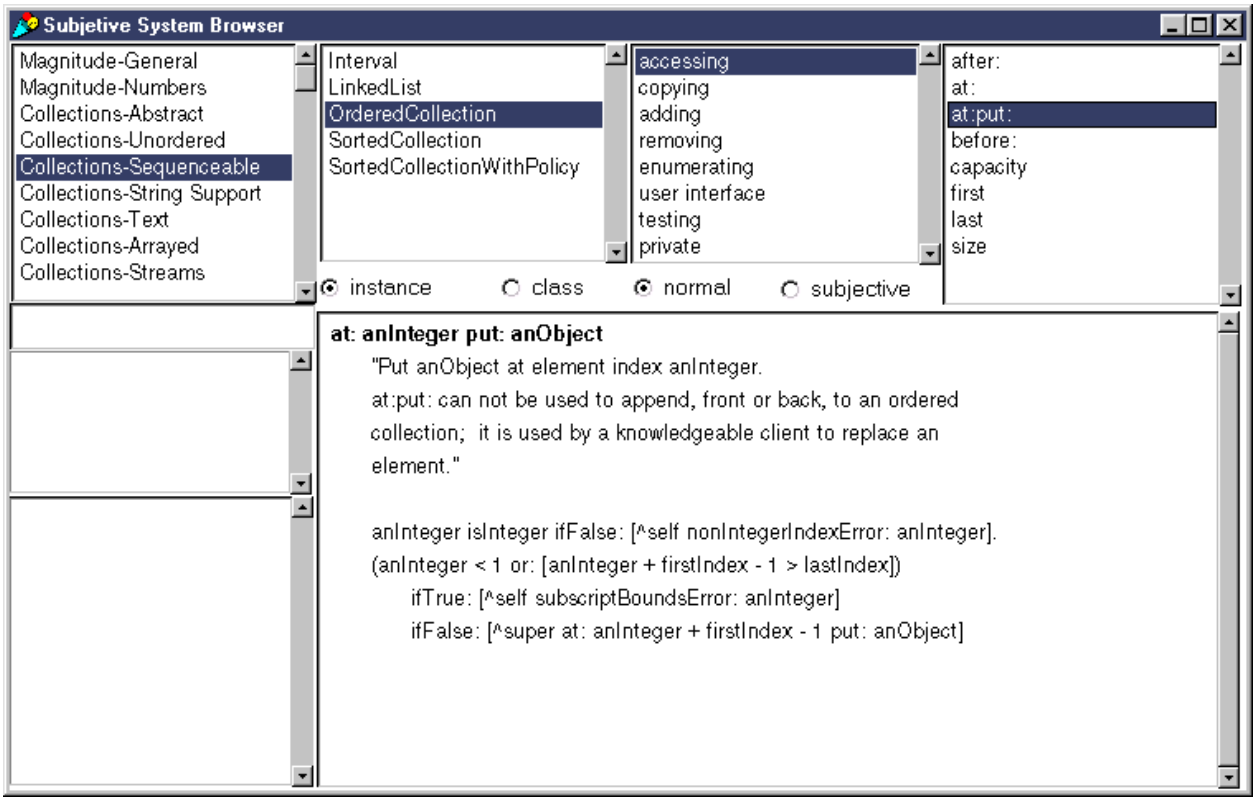


Figura 25: Manejo de clases y métodos "normales" dentro del Browser Subjetivo.

Manejo de la funcionalidad adicional que permita trabajar con Subjetividad.

Para esto alcanza con seleccionar el botón “subjective” ubicado debajo de la ventana de “categorías”. Como se puede observar en la siguiente imagen, el browser es prácticamente análogo al anterior, solamente que comienzan a utilizarse los paneles ubicados en la parte inferior izquierda del browser. Estos paneles, como veremos más adelante, nos permiten trabajar selectivamente con la lógica de fuerzas, las fuerzas y los determinantes de la clase seleccionada.

En el ejemplo, se muestra la clase “Stack” de nuestro caso de estudio. Se pueden observar las tres fuerzas existentes en el ambiente subjetivo “State”, “Sender” y “Content”, y los determinantes “Vacío”, “No Vacío” para la clase “Stack” y la fuerza “State”.

Para trabajar con el ambiente subjetivo, debe estar siempre seleccionado el botón “subjective”.

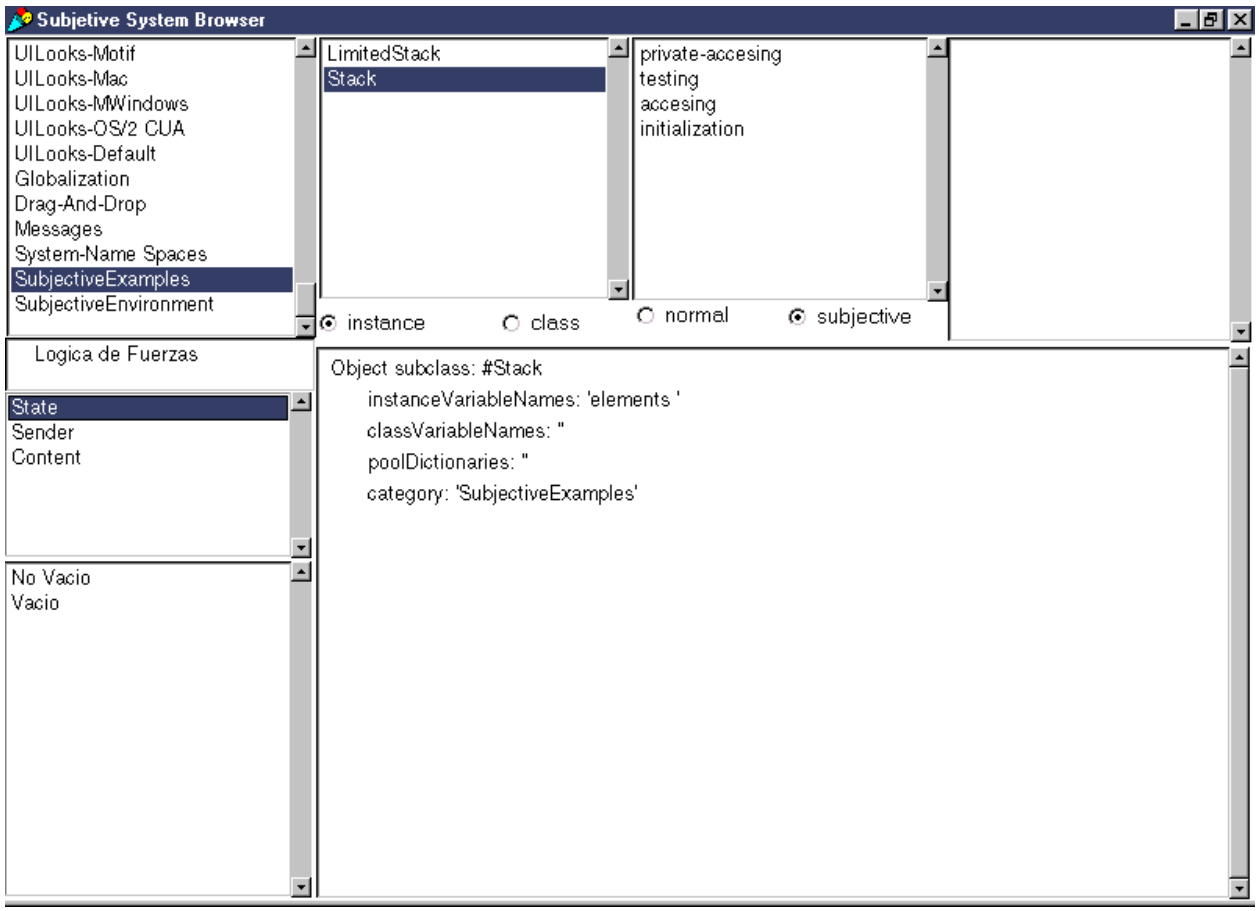


Figura 26: Manejo de clases "subjetivas" dentro del Browser Subjetivo, incluyendo sus métodos, lógica de fuerzas y determinantes.

Agregado, eliminación , modificación y obtención de las fuerzas existentes en el ambiente.

Como ya se explicó en la herramienta de “Administración de Fuerzas”, si uno se ubica en el panel

correspondiente a las fuerzas del ambiente, y presionando el botón derecho del mouse sobre el mismo, aparece un menú con la posibilidad de realizar Alta, Baja y Modificación de Fuerzas. Este menú invoca al "Administrador de Fuerzas" que vimos anteriormente.

Agregado, eliminación y obtención de los determinantes de una fuerza para una clase dada.

Cuando el programador determina que una clase es subjetiva, deberá realizar un análisis para identificar cuáles son los determinantes de cada fuerza que influyen a los objetos de la clase. Una vez en claro esta información, deberá ingresar los determinantes que surjan del análisis previo, para la fuerza y la clase en cuestión. Para esto, estando ubicado sobre la clase subjetiva que está trabajando, debe seleccionar la fuerza a la cual desea agregarle determinantes. Entonces sobre el panel de determinantes, dispone de un menú que le permite agregar y eliminar determinantes (opciones "add..."/"delete..." del menú desplegado con el botón derecho del mouse).

En el ejemplo siguiente, se muestra el browser, al momento de realizar el agregado del determinante "Intermedio" para la fuerza "State" y la clase "LimitedStack".

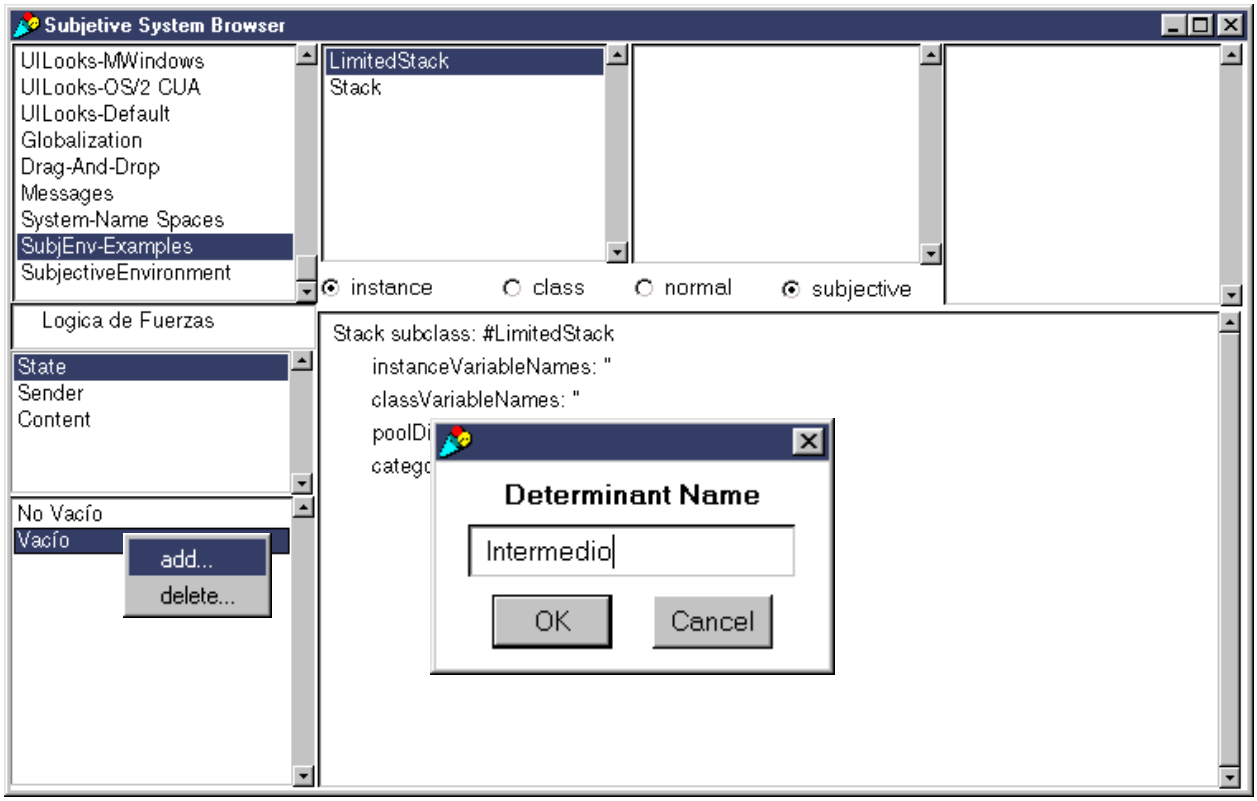


Figura 27: Agregado de determinantes para una clase "subjetiva" y una determinada fuerza dentro del Browser Subjetivo.

Obtención del código fuente de los métodos normales para una determinada clase.

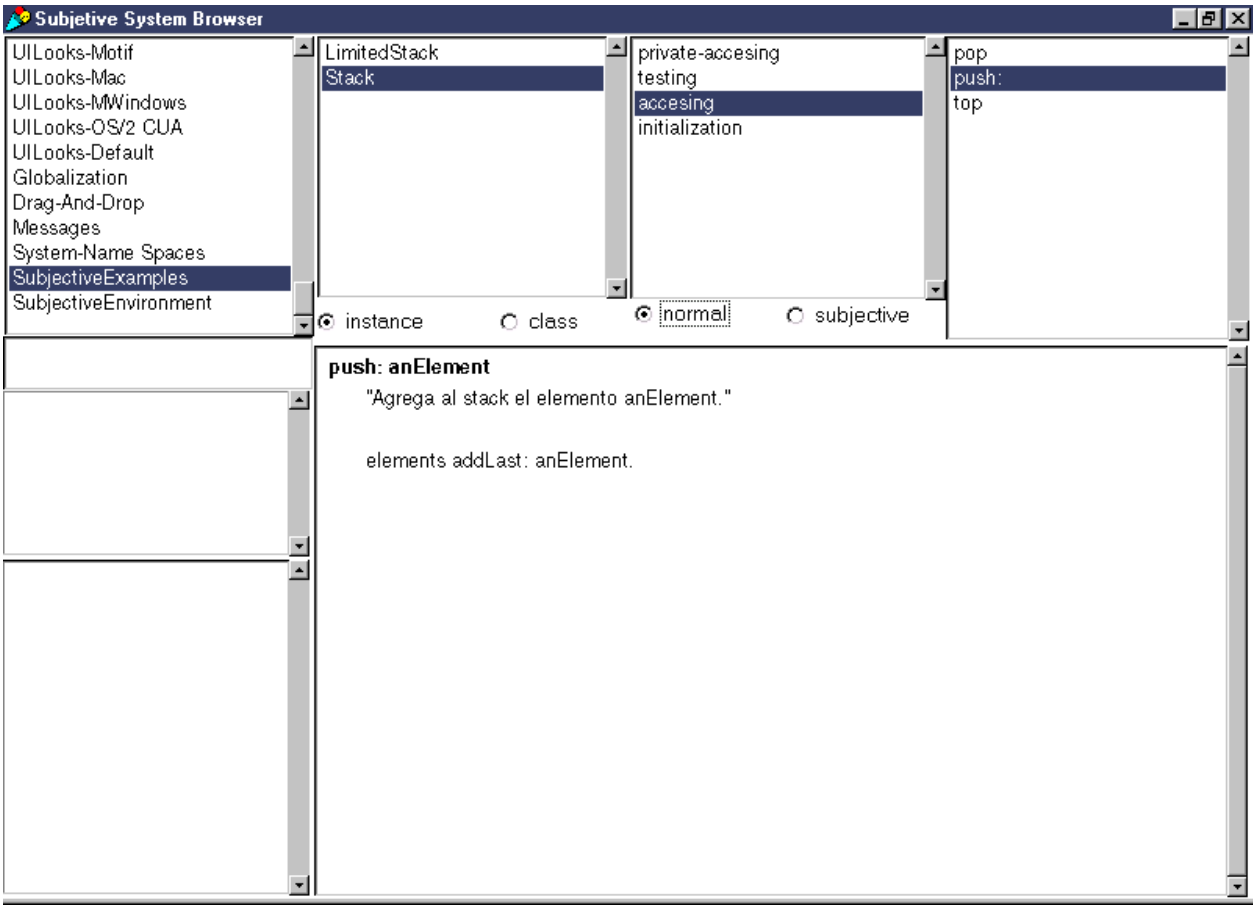
De la misma forma que en el browser tradicional, eligiendo el switch "normal", se puede editar, modificar

o definir por primera vez el código de los distintos métodos. También se puede reemplazar la implementación “subjetiva” de un mensaje por una implementación “normal”.

Aquí debemos recordar que la definición de mensajes “normales” no está necesariamente ligada a clases “normales”; una clase “subjetiva” puede tener métodos que no varíen su comportamiento.

En el siguiente ejemplo, podemos ver el método del mensaje “push:” definido en la clase “Stack”; en este caso, la clase “Stack” es subjetiva, pero el método “push:” se considera “normal” pues no varía su comportamiento de acuerdo con la cantidad de elementos del stack, como puede suceder con otros mensajes de esta clase.

Figura 28: Definición de un método "normal" para una clase "subjetiva" utilizando el Browser Subjetivo.



Si ocurre que se selecciona un mensaje no subjetivo estando seleccionado el botón de “subjective” entoces se muestra el aviso indicando que “El método tiene implementación normal” (es decir, no subjetiva) , en lugar de mostrar el código del método. Este hecho se puede apreciar en el siguiente ejemplo, donde vemos nuevamente el método del mensaje “push:” del ejemplo anterior. Aquí se ha seleccionado este método con el fin de visualizar su código fuente pero teniendo presionado el botón “subjective”. Recordemos que “push” tiene una implementación normal.

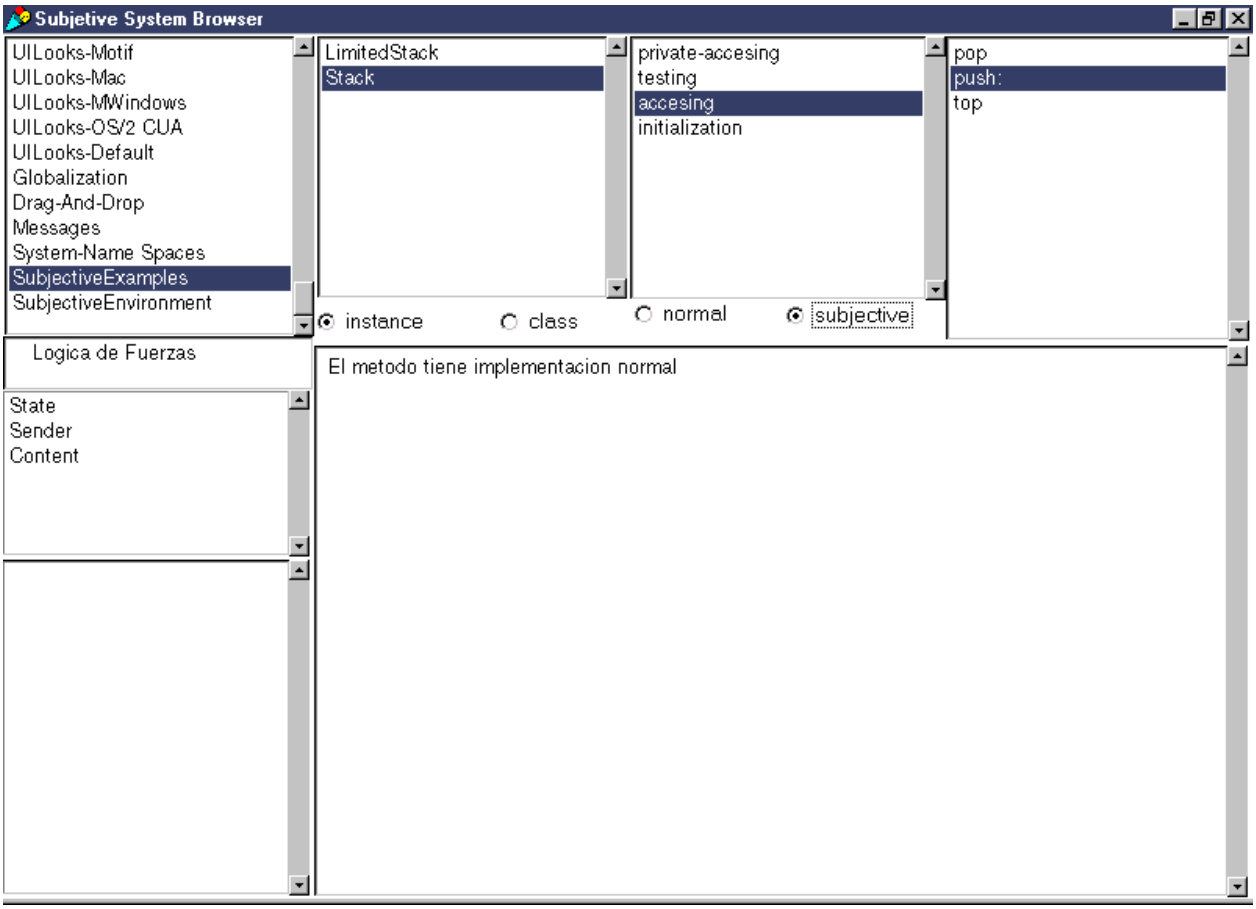


Figura 29 : Selección de un método normal dentro del Browser Subjetivo, suponiendo que el método es subjetivo (botón subjective)

Definición del código fuente de los métodos subjetivos para una determinada clase.

Similarmente al browser tradicional, eligiendo el switch "subjetivo", una clase, una fuerza y un determinante, se puede editar, modificar o definir por primera vez el código de los distintos métodos. También se puede reemplazar la implementación normal de un mensaje por una subjetiva.

En el siguiente ejemplo, podemos ver el método del mensaje "pop" definido en la clase "Stack"; en este caso, la clase "Stack" es subjetiva, y el mensaje "pop" también debido a que su comportamiento varía de acuerdo con la cantidad de elementos del stack. Por tal motivo a continuación se muestran ambas implementaciones, cada una bajo la fuerza y el determinante que le corresponde.

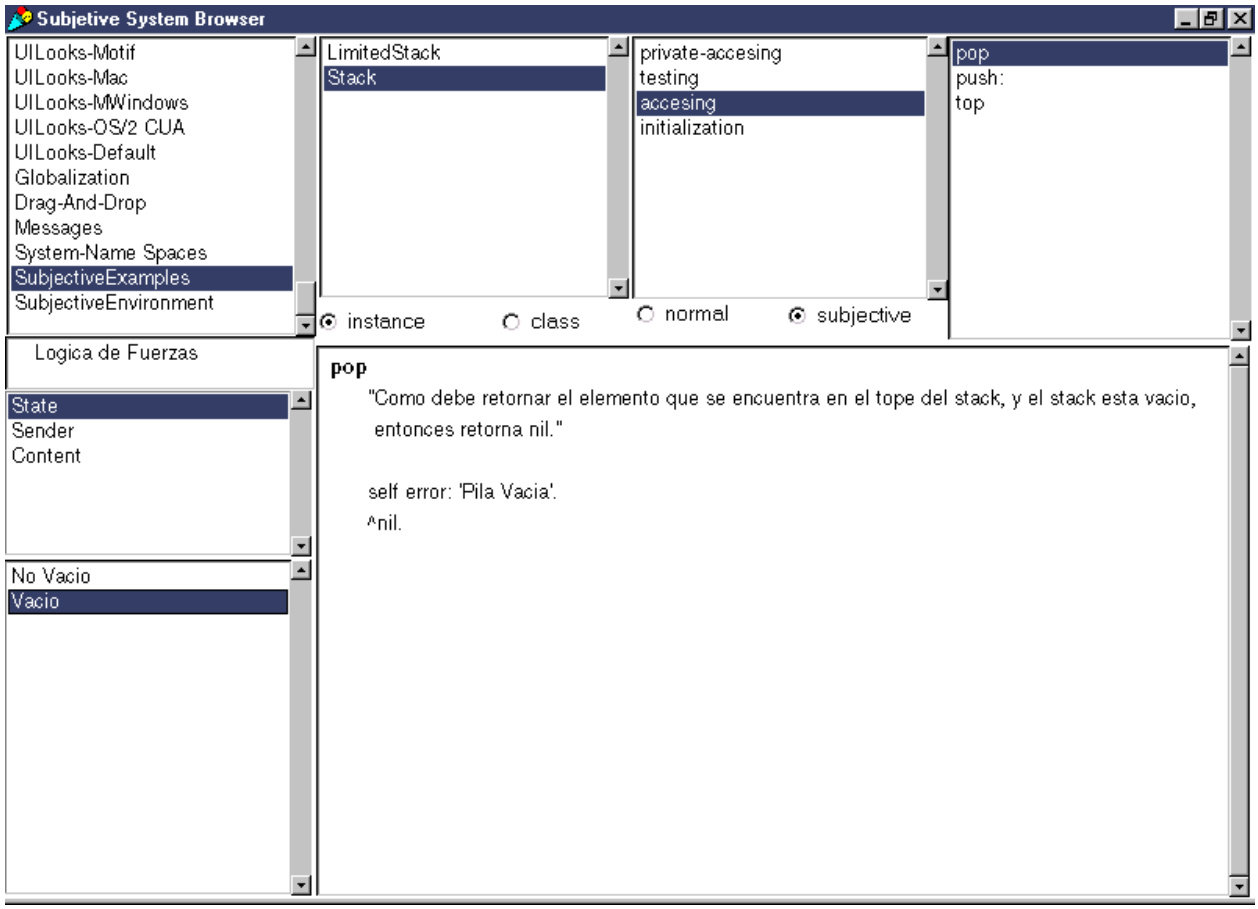


Figura 30: Definición de una de las implementaciones del mensaje subjetivo "pop" de la clase Stack. (Fuerza: State / Determinante: Vacío)

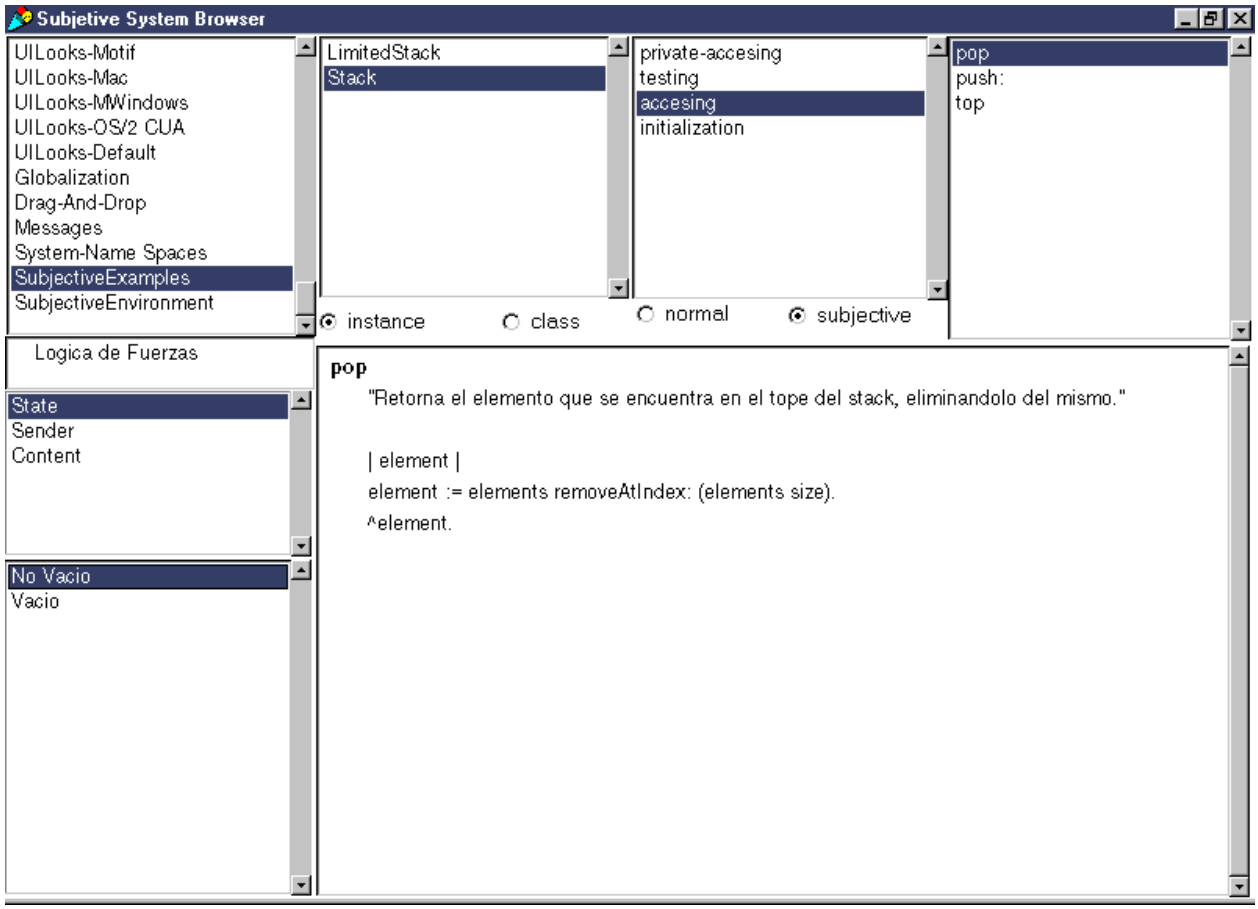


Figura 31 : Definición de una de las implementaciones del mensaje subjetivo "pop" de la clase Stack. (Fuerza: State / Determinante: No Vacío)

Si ocurre que se selecciona un mensaje subjetivo estando seleccionado el botón de “normal” entoces se muestra el aviso indicando que “El método tiene implementación subjetiva”, en lugar de mostrar el código del método. A continuación se muestra un ejemplo donde se ha seleccionado nuevamente el método del mensaje “pop:” del ejemplo anterior con el fin de visualizar su código fuente, pero teniendo seleccionado el botón de “normal”.

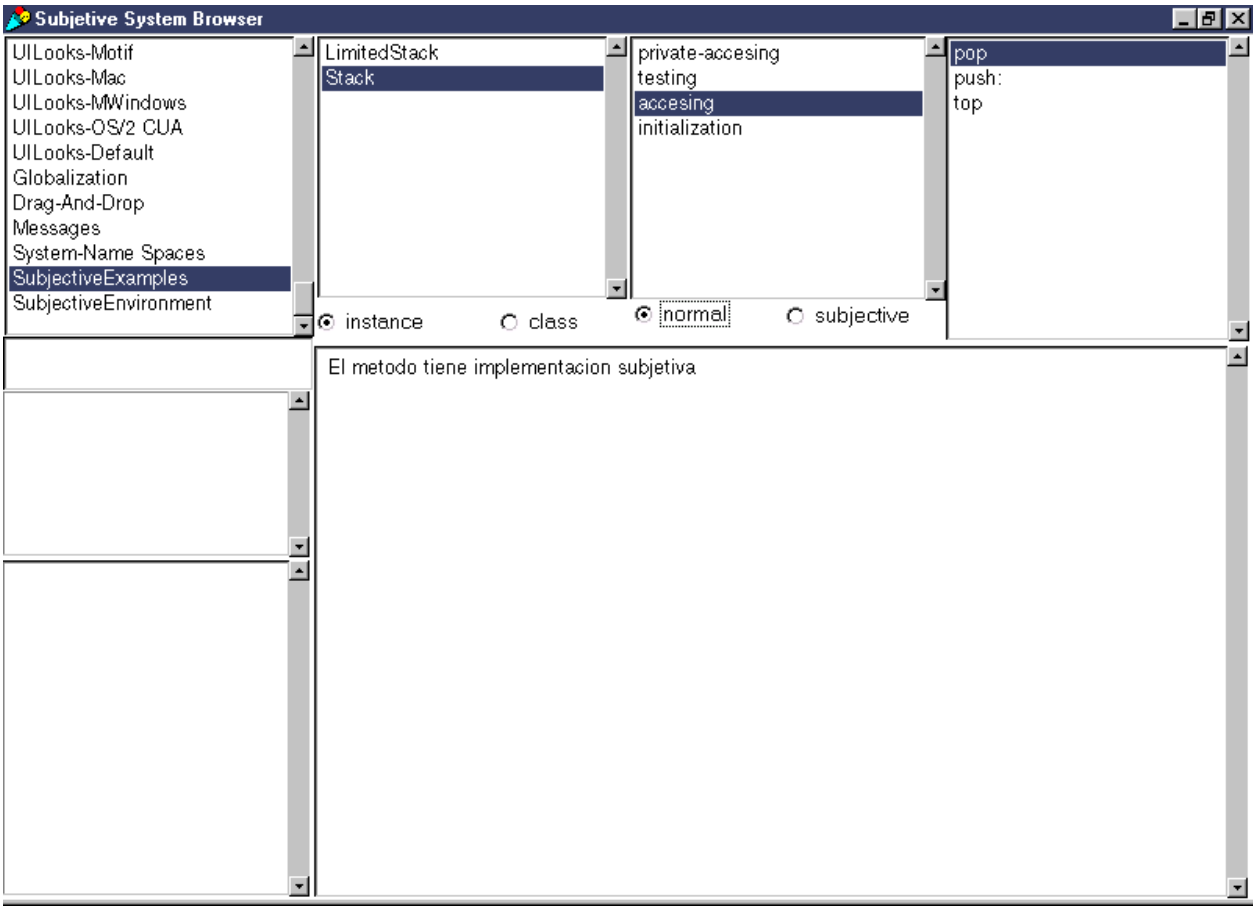


Figura 32 : Selección de un método subjetivo, suponiendo que el método es normal (botón normal)

Agregado y obtención de la Lógica de Fuerzas para una clase o método determinados.

En algún momento el programador necesita definir o modificar la lógica de fuerzas de una determinada clase subjetiva. Seleccionando el panel “Lógica de Fuerzas” podrá realizar esta tarea.

Como explicáramos en la parte teórica, puede suceder que esta lógica necesite estar definida a nivel de clase o de método (de la clase subjetiva en cuestión). Si desea definirla a nivel de método, el método asociado deberá estar seleccionado en el browser; si no existe ningún método seleccionado, se asume que la definición de lógica de fuerzas que se está realizando es a nivel de clase.
En caso que no exista definida la lógica de fuerza, se mostrará una plantilla que le indicará al programador el formato que debe poseer dicho bloque.

En la sección de Lógica de Fuerzas ya hemos analizado en detalle el bloque del lógica de fuerzas, por este motivo, aquí nos limitaremos a mostrar dos ejemplos relacionados con nuestro caso de estudio.
En la primer pantalla podemos observar que la clase subjetiva “Stack” no posee una lógica de fuerzas a nivel de clase, debido a que sin seleccionar un método en particular, la lógica de fuerzas muestra simplemente la plantilla indicativa para el programador.

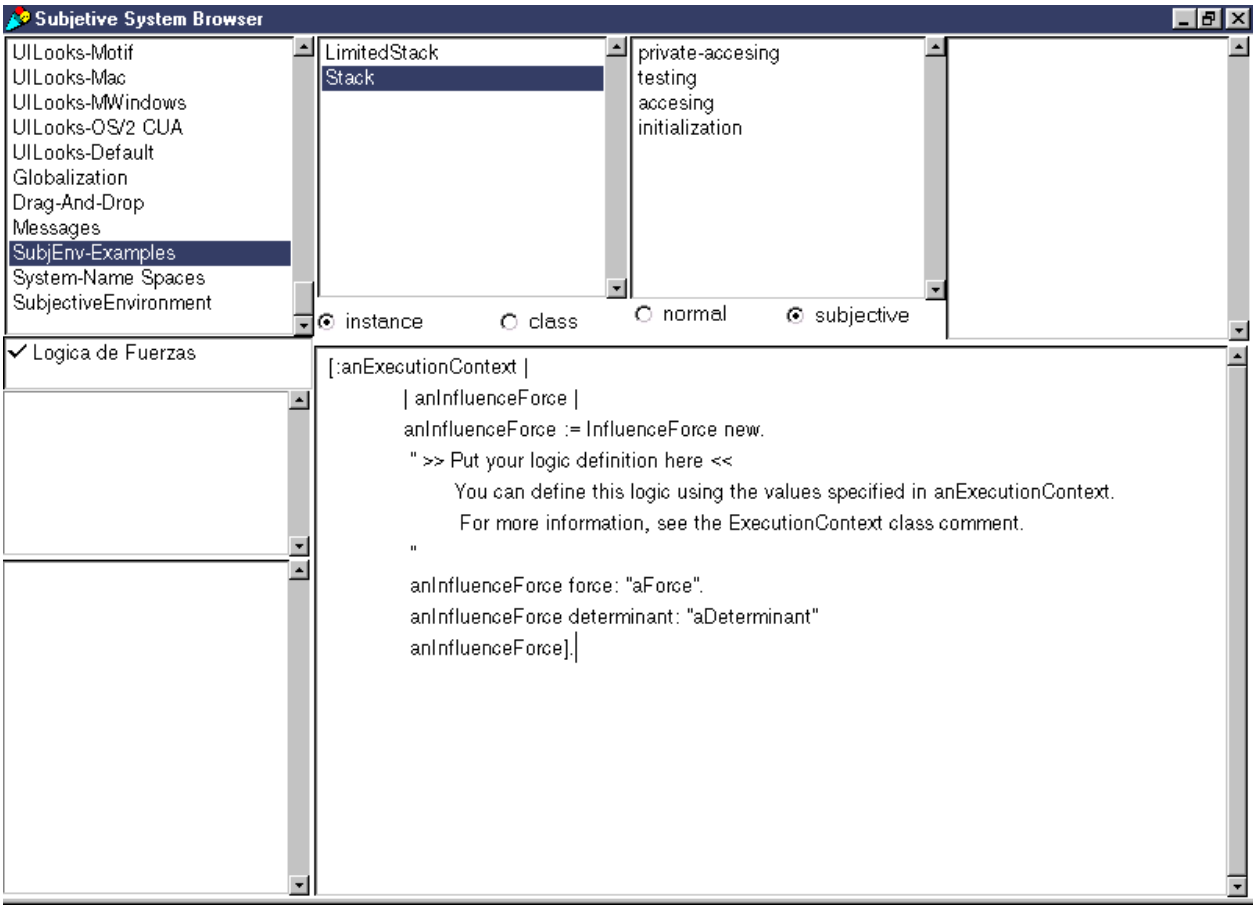


Figura 33: Bloque de Lógica de Fuerzas a nivel de clase (plantilla)

En la segunda pantalla podemos observar la definición de lógica de fuerzas para el método “pop”. Aquí cabe mencionar, que si bien esta lógica se definió a nivel de método, nada nos hubiera impedido (y de hecho, para este ejemplo, el comportamiento obtenido hubiese sido el mismo) definir la lógica a nivel de clase; simplemente se realizó de esta manera para mostrar ambas posibilidades.

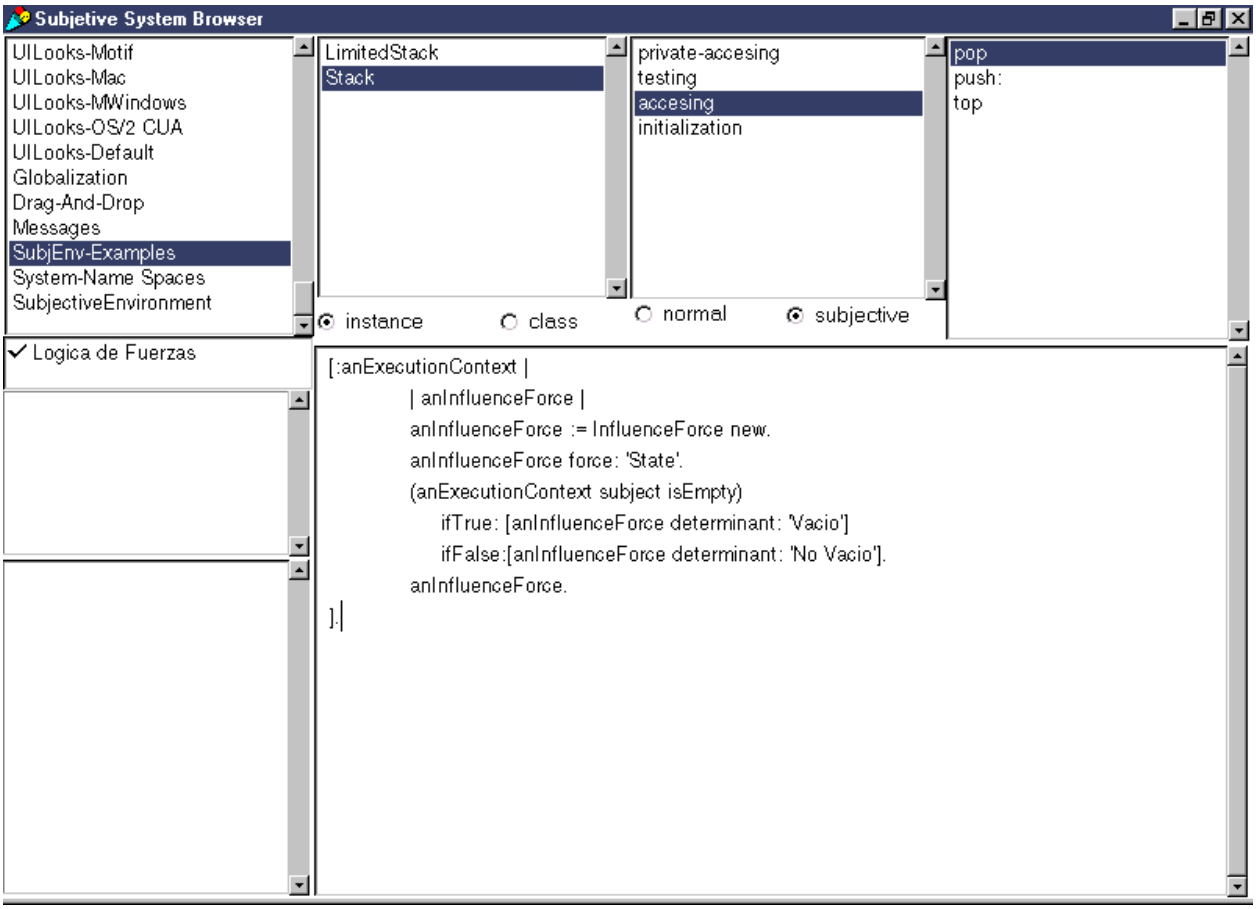


Figura 34 : Bloque de Lógica de Fuerzas a nivel de método (lógica ya implementada)

Funcionamiento interno del Browser del ambiente subjetivo

Veamos ahora qué pasos sigue el browser subjetivo cuando necesita compilar y grabar:

a. La lógica de fuerzas
b. Un método “normal”
c. Un método “subjetivo”

Compilación y almacenamiento del bloque de Lógica de Fuerzas

Similarmente al browser tradicional, se compila el texto ingresado para chequear los errores sintácticos; en caso que haya errores despliega los mensajes correspondientes; sino guarda el texto ingresado para la lógica de fuerzas (ya sea a nivel de clase/método) en el objeto SubjectivitySmalltalk (en realidad en la variable de instancia “forceLogicDefinition” de este objeto). En la sección “La Lógica de Fuerzas” se analiza la clase “ForceLogicDefinition”, explicando cómo se almacenan los bloques de lógica de fuerzas.

Compilación y almacenamiento de un método "normal" para una determinada clase

De la misma forma que en el browser tradicional, se compila el texto ingresado, desplegando los errores sintácticos (si los hubiese) o guardando el método compilado (instancia de la clase "CompiledMethod") resultante de la compilación del texto, en el diccionario de métodos de la clase asociada ("MethodDictionary").

Compilación y almacenamiento de un método "subjetivo" para una clase y mensaje determinados

Antes que nada debe compilarse el texto ingresado, para desplegar los errores sintácticos (si los hubiese).
Si el texto no tiene errores, debemos proceder a almacenarlo.

En caso de tratarse de la primera implementación de un mensaje subjetivo, deberá crearse el "SubjectivityCompiledMethod" para luego poder almacenar el método compilado bajo la fuerza y el determinante que corresponda dentro del "SubjectivityCompiledMethod". En caso que ya exista otra implementación, este "SubjectivityCompiledMethod" ya estará generado en el diccionario de métodos de la clase, y simplemente se procederá a agregar el nuevo método compilado bajo la fuerza y el determinante correspondiente.

(Recordar que el "SubjectivityCompiledMethod" generado por el browser posee el código necesario para derivar la ejecución al método compilado resultante de la compilación del texto ingresado).

El Launcher Subjetivo (Clase "SubjectivityVisualLauncher")

Otra herramienta que fue necesario modificar, fue el launcher (clase "VisualLauncher"). Para ello se creó la clase "SubjectivityVisualLauncher", subclasificándola de la clase "VisualLauncher". La nueva clase permite crear launcher's subjetivos, o sea: que mantienen una coherencia con respecto al ambiente subjetivo. Esto implica que manejan browsers subjetivos, agregan opciones de menú como la apertura de la herramienta que permite administrar las fuerzas, etc.

La creación del primer launcher subjetivo se lleva a cabo durante el proceso de instalación del nuevo ambiente cuando, en un determinado momento, se solicita ejecutar:

SubjectivityVisualLauncher open.

Y luego se solicita cerrar el launcher actual (el laucher "normal") puesto que de otra forma se generarían inconsistencias en el manejo de algunas objetos.

Los Inspectors

Los inspectors no fueron modificados en esta implementación, ya que la funcionalidad tradicional se mantuvo, y el resto de la funcionalidad que debía extenderse se puede realizar a través del Debugger. De todos modos, puede considerarse como un futuro trabajo de esta tesis.

El Debugger (Clases “Debugger” y “SubjectivityDebugger”)

Una herramienta indispensable que fue necesario modificar fue el debugger. Para ello se creó la clase “SubjectivityDebugger”, subclasificándola de la clase “Debugger”. La nueva clase representa al debugger del ambiente subjetivo, y su objetivo es proporcionar al programador los mecanismos necesarios para que éste pueda realizar el seguimiento de un proceso suspendido.
Este proceso puede ser normal o subjetivo, en cuyo caso muestra la fuerza y el determinante en la que fue definido el método.

El debugger subjetivo permite visualizar el código del método (ya sea normal o subjetivo), o la lógica de fuerzas, modificarlos y reiniciar la ejecución una vez aceptadas las modificaciones.

A continuación se detallan las funcionalidades agregadas a esta clase.

Obtención de la fuerza y el determinante de un método subjetivo

Al seleccionar el contexto correspondiente a un método subjetivo, despliega la fuerza y el determinante en la que fue definido el método.

Para ver un ejemplo de esto retomaremos la clase “Stack” de nuestro caso de estudio y haremos lo siguiente. En el método “pop” definido para la fuerza “State” y el determinante “No Vacío”, invocaremos un método inexistente “errorMessageForDebuggingTest”, simulando un error del programador, lo cual generará un error de ejecución que invocará al debugger.

O sea que la definición errónea de este método quedaría de la siguiente forma:

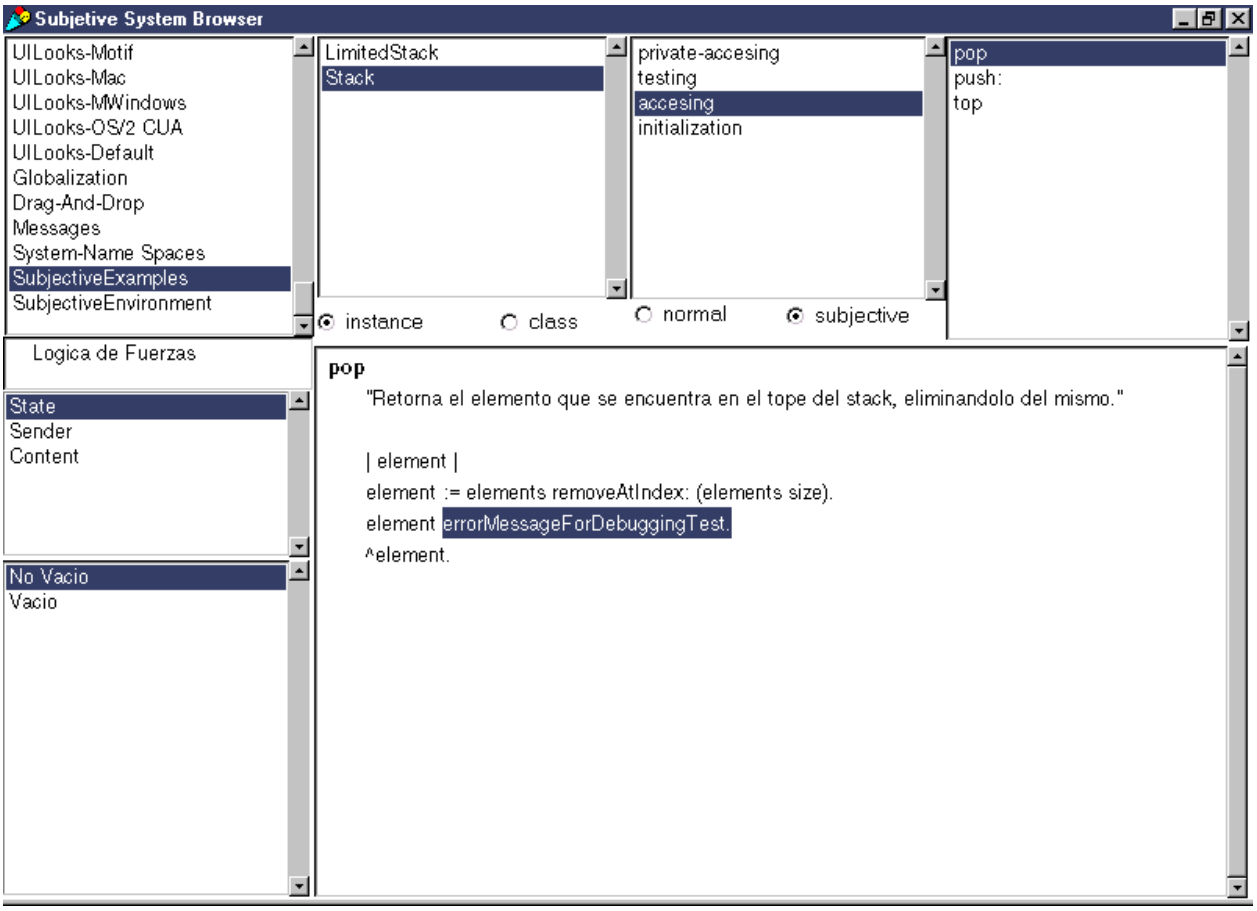


Figura 35: Generación de código erróneo en el método "pop" para forzar la invocación del Debugger Subjetivo en tiempo de ejecución.

Lo único que falta ahora es ejecutar algún código que realice una invocación al mensaje “pop” para un stack con elementos. Entonces utilizando el transcript, realizamos un “Do-It” sobre el siguiente código:

```
| s  |
s:= Stack new.
s push: (String new).
s pop.
```

Al llegar al método “pop” se produce el error esperado, pudiendo de esta forma abrir el debugger subjetivo. Desplazándonos hasta la segunda línea, vemos la siguiente situación:

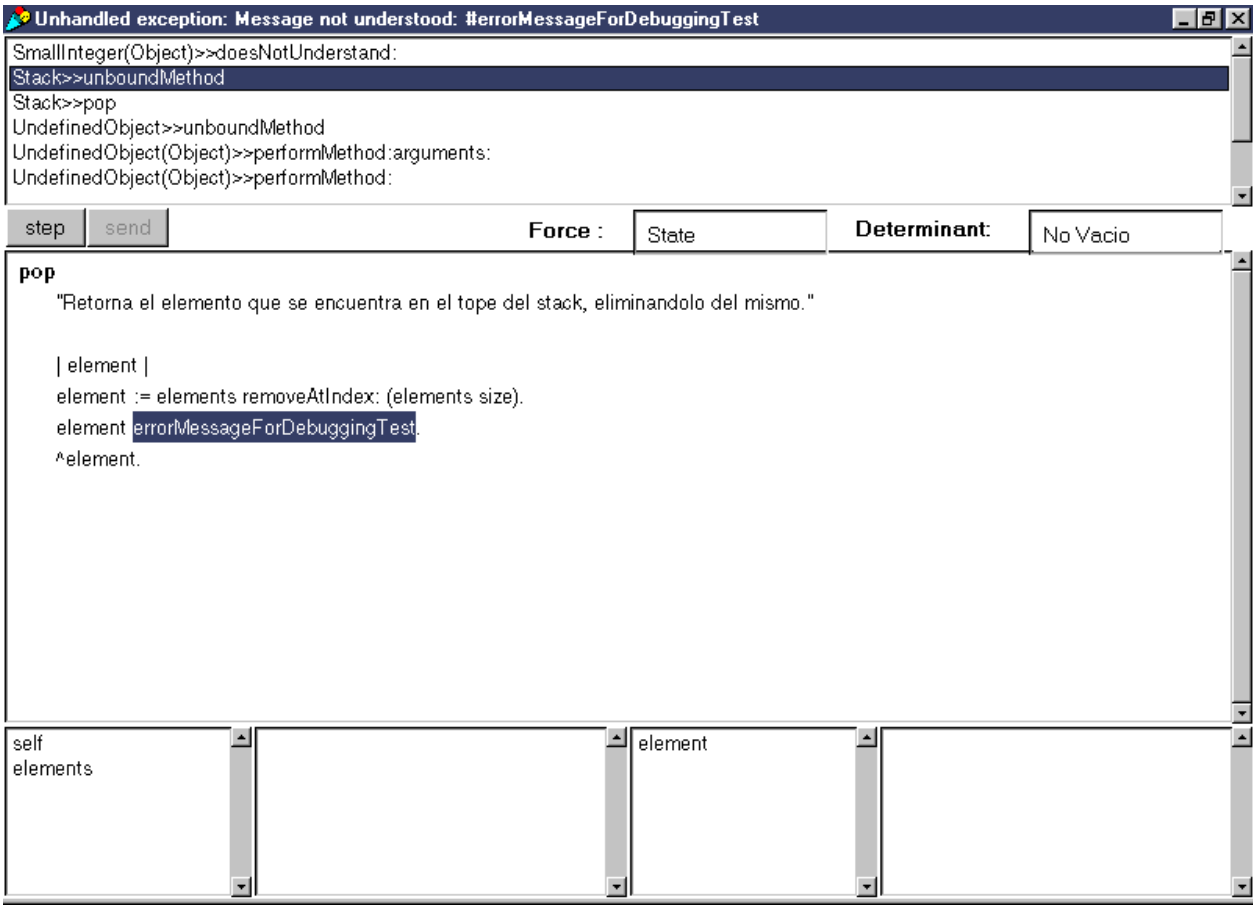


Figura 36: El Debugger Subjetivo ante un error producido en un método subjetivo.

El mensaje “errorMessageForDebuggingTest” no se pudo comprender, y por eso el debugger subjetivo arroja el error “Unhandled exception: Message not understood: #errorMessageForDebuggingTest”. Como podemos observar en el debugger, el método “pop” donde se produjo el error es el definido en la fuerza “State”, determinante “No Vacío”.

Compilación y grabación del método normal en la clase correspondiente y reiniciado de la ejecución

De la misma forma que en el debugger tradicional, permite editar el código de un método normal, aceptarlo y reiniciar la ejecución desde ese punto.

Compilación y grabación del método subjetivo en el SubjectivityCompiledMethod correspondiente y reiniciado de la ejecución

Similarmente al debugger tradicional permite la edición de un método subjetivo, aceptarlo y reiniciar la ejecución desde ese punto. Notar que la decisión de la lógica de fuerzas ya fue realizada; por lo tanto, la fuerza y el determinante no varían.

Compilación y grabación del bloque de lógica de fuerzas y reiniciado de la ejecución

Similarmente al debugger tradicional, permite editar el bloque de lógica de fuerzas, aceptarla y reiniciar la ejecución desde ese punto.

Para ver un ejemplo de esto retomaremos nuevamente la clase “Stack” de nuestro caso de estudio y haremos lo siguiente. En la lógica de fuerzas definida a nivel de la clase “LimitedStack” introduciremos la invocación a un método inexistente “errorMessageForDebuggingTest”, con el fin de generar un error de ejecución al evaluar el bloque de lógica de fuerzas, y poder de esta manera acceder al debugger. O sea que la definición errónea de esta lógica de fuerzas quedaría de la siguiente forma:

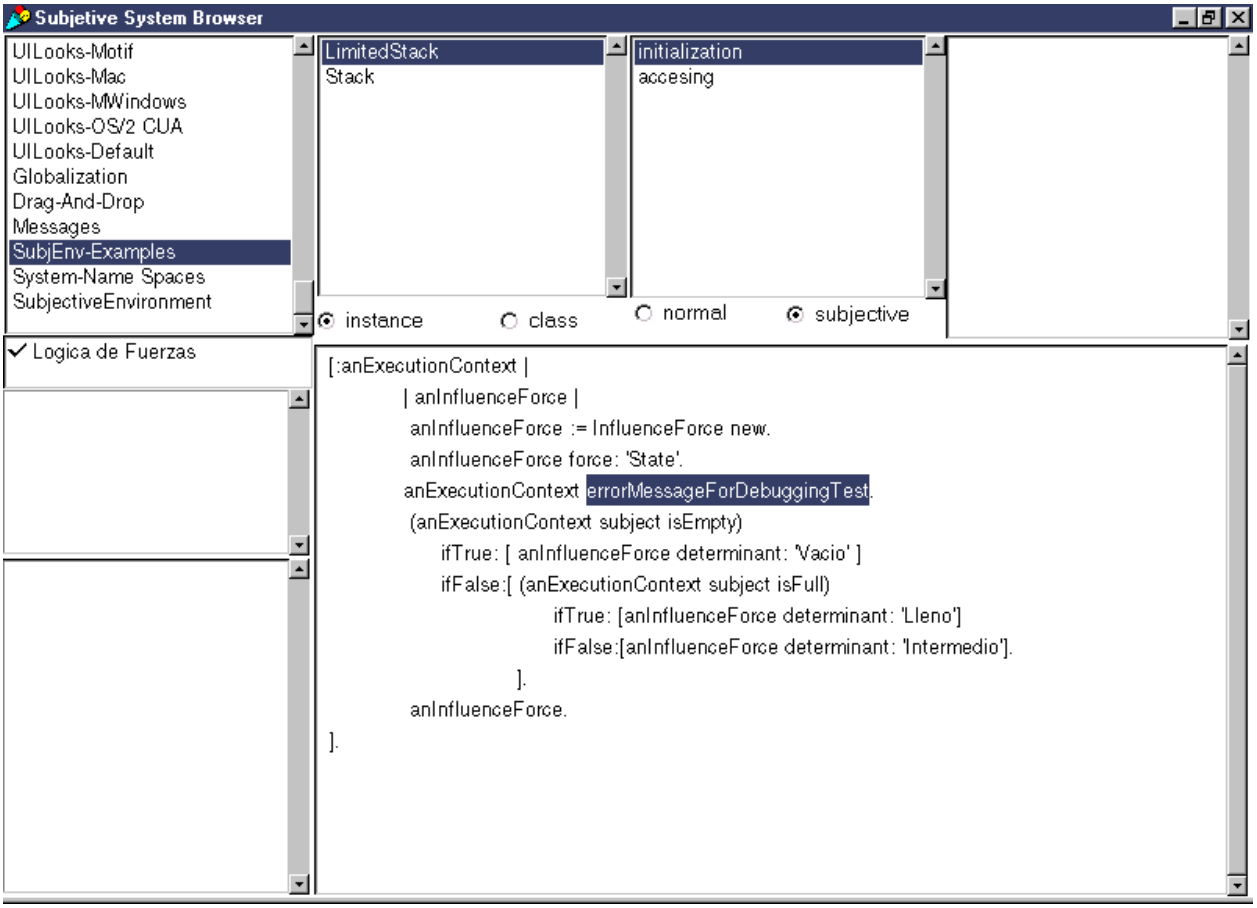


Figura 37: Generación de código erróneo en un bloque de lógica de fuerzas para forzar la invocación del Debugger Subjetivo en tiempo de ejecución.

Lo único que falta ahora es ejecutar algún código que realice una invocación a un mensaje subjetivo para que se evalúe el bloque de lógica de fuerzas y se produzca el error. Entonces utilizando el transcript, realizamos un “Do-It” sobre el siguiente código:

```
| s   |
s:= LimitedStack new.
s push: (String new).
```

Al llegar a la lógica de fuerzas del “LimitedStack” se produce el error esperado, pudiendo de esta forma abrir el debugger. Desplazándonos hasta la segunda línea, vemos la siguiente situación:

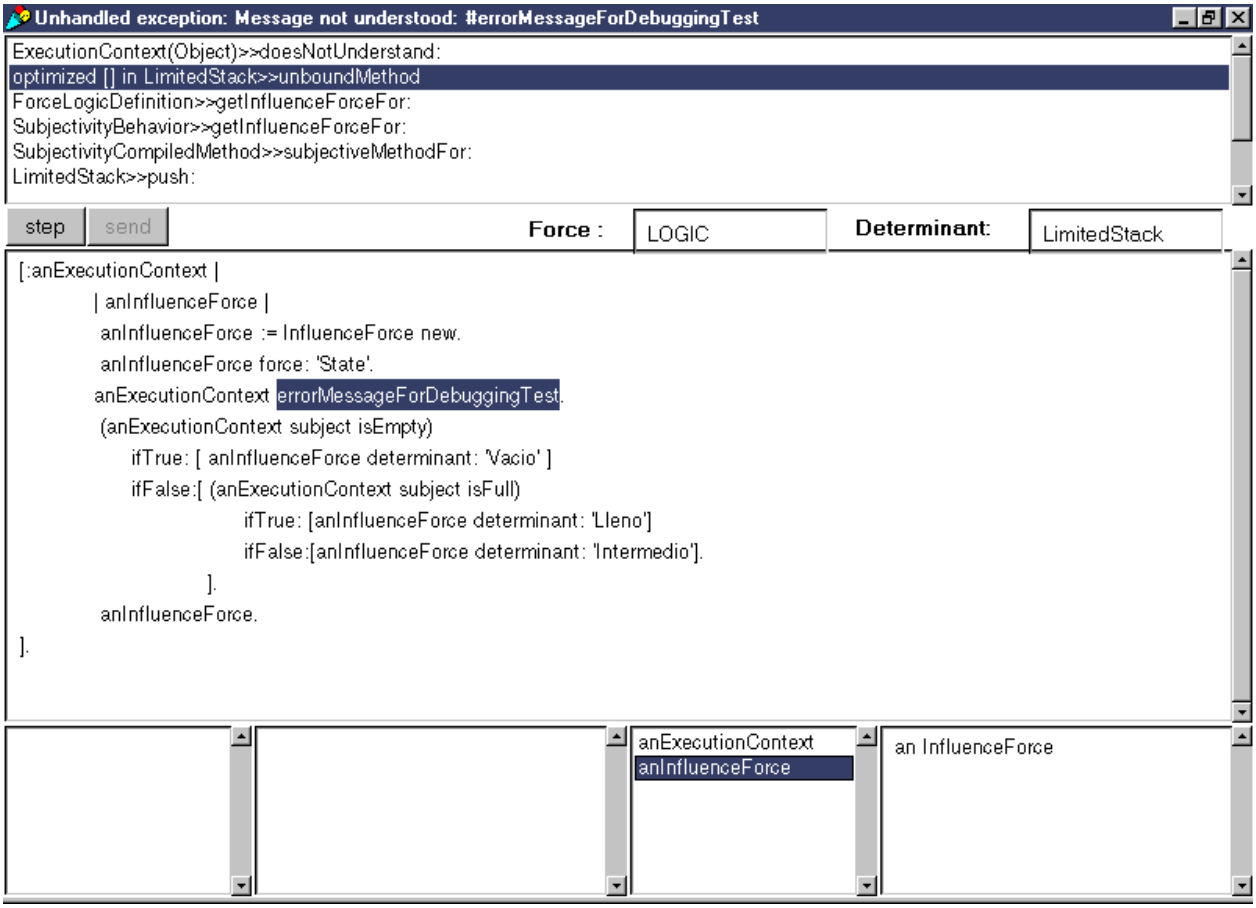


Figura 38 : El Debugger Subjetivo ante un error producido en la lógica de fuerzas.

Aquí se puede modificar el bloque de lógica de fuerzas, cambiar los valores retornados por la evaluación del mismo -esto se puede hacer modificando los valores que tiene el objeto “anInfluenceForce”-, o cambiar los valores de las variables de instancia de “anExecutionContext”. De esta manera se podría manipular el objeto stack -que es el “subject” del objeto “anExecutionContext”- con el fin de modificar su estado interno, para que la fuerza y/o determinante resulten diferentes.

Modificación del Código Correspondiente al SubjectivityCompiledMethod o el NullCompiledMethod

El Debbugger subjetivo permite visualizar el código fuente correspondiente a los bytecodes del SubjectivityCompiledMethod o de un NullCompiledMethod (ver secciones “El método Nulo” y “El SubjectivityCompiledMethod”). Sin embargo, no permite la modificación de estos códigos, ya que una modificación errónea podría ocasionar que deje de funcionar el ambiente subjetivo.

Conclusiones

La “Subjetividad” es un área de interés e importancia creciente en la comunidad orientada a objetos. Especialmente en sistemas de gran envergadura o en conjuntos de aplicaciones, es muy importante que un mismo objeto parezca y se comporte diferente cuando se lo utiliza en distintos contextos, por distintos clientes, de diferentes maneras, en tiempos distintos. El soporte de tal subjetividad promete gran flexibilidad al momento de escribir, extender y combinar sistemas y aplicaciones orientadas a objetos, al mismo tiempo que genera muchas cuestiones técnicas y filosóficas interesantes.

Comparaciones entre “Nuestro Enfoque de la Subjetividad” vs. el “Enfoque de la Subjetividad de otros autores”

A continuación realizaremos algunas comparaciones entre nuestro enfoque sobre el tema de la “Subjetividad” y el enfoque realizado por otros autores con respecto a este tema. Para ello tomaremos como referencia los trabajos que consideramos más importantes desde el punto de vista comparativo con nuestra tesis.

Sobre el modelo de modos propuesto por Taivalsaari

El planteo realizado por Antero Taivalsaari en su paper denominado “Object-oriented programming with modes” [TAI/93], es muy interesante ya que se puede trazar una analogía entre su postura y lo que para nosotros sería la “Fuerza de Estado”.

Los “modos” (o “estados lógicos”) que él define, son análogos a los “determinantes” de nuestra “fuerza de estado” (de hecho plantea el mismo ejemplo del stack, donde los “modos” podrían ser “vacio”, “lleno”, etc.).

Para cada “modo” define un conjunto de operaciones, como nosotros definimos una implementación diferente para cada determinante de la fuerza predominante. También deja en claro la necesidad de poder definir operaciones que sean “independientes de los modos”, que es exactamente el motivo por el cual nosotros planteamos un ambiente mixto, puesto que no todas las operaciones tienen por qué comportarse diferente en diferentes modos; estas operaciones deben poder ser definidas una sola vez, y en forma independiente de los “modos” que maneje el objeto.

Sus “funciones de transición” son análogas a lo que nosotros denominamos “lógica de fuerzas”, ya que las funciones de transición, se utilizan para determinar cuál es el siguiente “modo” del objeto, y la lógica de fuerzas se utiliza para obtener cuál es el determinante actual del objeto, bajo la fuerza de estado. La diferencia entre estos dos mecanismos, es que las funciones de transición se aplican cuando se finaliza la ejecución del conjunto de operaciones asociado al mismo, pasando el objeto al siguiente modo. En cambio, la lógica de fuerzas se aplica al recibir un mensaje, y es en ese momento cuando se deduce el determinante.
Por ejemplo, si la pila se encuentra en modo “vacío” y recibe un mensaje “push”, la función de transición determinará que la pila pasa al modo “no vacío”. Si luego recibe un mensaje “pop”, como se encuentra en el modo “no vacío”, devuelve el elemento y pasa al modo “vacío” nuevamente.
En cambio, al recibir un mensaje “push” como no es subjetivo, no hace consideraciones. Si luego recibe un mensaje pop, la lógica de fuerzas ve que la pila tiene un elemento, por lo tanto el determinante es “no vacío”.

Parece ser que, aún con la presencia de “modos” no pueden evitarse las sentencias condicionales, y se necesitará recurrir a sentencias “if-case” para determinar el “modo”. A primera vista esto refutaría la idea

de introducir “modos” a los lenguajes de orientación a objetos, pero en realidad esto no resulta un problema serio ya que existen soluciones elegantes, como las “funciones de transición”. Las sentencias condicionales para la determinación del “modo” se encuentran solamente en la “función de transición” y no en cada una de las operaciones. Lo mismo ocurre en nuestra lógica de fuerzas. Además, otra analogía, es que las “funciones de transición” no pueden ser invocadas explícitamente sino que son manejadas automáticamente por el sistema, ya que no forman parte de la interface pública del objeto.
La idea de utilizar “funciones de transición” en la programación orientada a objetos no es totalmente nueva; ya había sido adoptada previamente por Pernici en el “modelo de roles”.

Otro punto en el que coincidimos completamente con el autor es el que se refiere al tema de descripción del comportamiento. El hecho que el programador deba explícitamente describir el comportamiento de una operación para cada modo diferente, parece ser, a primera vista, una molestia. Pero en realidad creemos que resulta una ventaja. El enfoque dado utilizando “modos” requiere que el programador piense acerca del comportamiento de una abstracción en cada situación posible. Esto sin duda requiere un mayor esfuerzo de codificación, pero sin dudas resultará en un código más seguro y con una distancia más estrecha entre el diseño y la implementación.

Una diferencia importante entre el trabajo de Antero Taivalsaari y el nuestro, es que él solamente contempla lo que para nosotros sería la “Fuerza de Estado”, y de hecho lo hace en una forma bastante parecida a la nuestra. Pero no contempla ni hace mención sobre otras posibilidades como el manejo de los “emisores” de los mensajes, o “contenidos” de los parámetros al invocar los mismos. Además se trata de una definición teórica de la cual no pudimos conocer si se llegó a implementar en forma práctica, y en tal caso de qué forma.

Sobre el modelo de sujetos propuesto por Harrison y Osher

La principal similitud de nuestro enfoque con el de Harrison y Ossher es que coincidimos en la naturaleza de la Subjetividad: los objetos varían su comportamiento de acuerdo con los diferentes factores que los influyen.

Una de las principales diferencias entre ambos enfoques es que nuestro punto de partida para desarrollar el tema de Subjetvidad fue modelar en forma natural el comportamiento complejo de los objetos del mundo real a partir de la observación de los mismos. En cambio, el punto de partida de Harrison y Ossher fue facilitar el desarrollo y la evolución de conjuntos de aplicaciones cooperativas.

Entendemos que en caso que se realice un desarrollo de aplicaciones en forma descentralizada como proponen Harrison y Ossher, debe tenerse la precaución de utilizar las herramientas adecuadas para evitar duplicar código, esfuerzo y trabajo. Uno debe obtener beneficios del trabajo descentralizado y no ganar problemas (por ejemplo: utilizando herramientas como “ENVY Developer”, etc.)

Haciendo una analogía con nuestro modelo, el significado de las reglas de composición de Harrison y Ossher es similar al de nuestra *lógica de fuerzas*, es decir en ambos casos se especifica de qué manera va a trabajar la subjetividad (en cierta forma es “la inteligencia” de la subjetividad). La gran diferencia es que las reglas de composición quedan fijas al momento de compilar, en cambio la lógica de fuerzas permite realizar cambios en forma dinámica durante el tiempo de ejecución.

Otra diferencia importante entre ambos enfoques es que nuestra visión *extiende* la Orientación a Objetos, mientras que la propuesta de Harrison y Ossher *hace uso* de la misma: el modelo clásico de objetos es el modelo de objetos visto por cualquier sujeto. La Programación Orientada a Sujetos incluye a la Programación Orientada a Objetos como uno de sus elementos tecnológicos.

Por último, en el modelo de Harrison y Osher se trabaja sobre las “vistas de los objetos”, o sea: cómo una aplicación (u otro objeto en general) ve a un objeto dado, y por lo tanto, la diferencia de comportamiento está dada por el comportamiento del objeto en cada una de las vistas; por ejemplo: cómo el pájaro y el leñador ven al árbol. En cambio, en nuestro modelo nos concentramos en el

comportamiento del objeto receptor del mensaje; el emisor del mensaje no se preocupa por cómo resuelve el mensaje dicho objeto, sino que el objeto receptor es el que define cómo se realiza este comportamiento.

Sobre el modelo de Roles propuesto por Pernici

Una interesente ventaja que presenta Pernici, coincidente con la postura de Taivalsaari y la nuestra, es que mediante la utilización de roles, el diseñador puede concentrarse en trabajar con un rol a la vez, permitiendo que se medite acerca del comportamiento de una abstracción en cada situación posible. Este no es el caso de la Programación Orientada a Objetos convencional en la cual el diseñador puede facilmente olvidarse de incluir ciertos testeos condicionales en las operaciones.

Como en el modelo de Harrison y Ossher, se puede trazar una analogía entre nuestra *lógica de fuerzas* y las reglas asociadas cada rol en el modelo de Roles de Pernici. Las reglas describen el comportamiento de un objeto dentro del rol, y la lógica de fuerzas también gobierna el comportamiento del objeto.

El diagrama de estados planteado en este modelo se podría aplicar para modelar nuestra fuerza de estado, aunque no es aplicable para las otras dos fuerzas definidas ya que no tiene sentido hablar de trancisiones entre distintos emisores.

Desde el punto de vista del análisis, los roles pueden ser útiles para identificar comportamiento subjetivo.

Sobre el modelo de Roles propuesto por Kristensen

Según el concepto de roles de Kristensen, un objeto se comporta de una forma esperada por un conjunto de otros objetos. De aquí se desprende que en nuestra aproximación de la Subjetividad, un rol podría ser implementado bajo la fuerza de emisor, y los determinantes serían los distintos conjuntos de objetos que esperan que el objeto se comporte de cierta forma, o sea: los sujetos. También, podría ser implementado bajo la fuerza de estado y cada uno de los roles (Oficinista, Pariente, Conductor, etc) sería un determinante.
Si bien el alcance de este trabajo no incluye estas pruebas, sería interesante intentar implementar el modelo propuesto por Kristensen de acuerdo a la analogía planteada previamente para verificar que esto sea así.

"Vistas", "Roles" y "Modos"

Estos enfoques poseen algunas similitudes y diferencias con respecto a lo planteado en nuestra tesis.

Un punto en común que poseen todas estas extensiones es que permiten a los objetos tener múltiples interfaces. Esto es diferente a los lenguajes convencionales de orientación a objetos, donde cada objeto presenta solamente una interface a sus clientes. Sin embargo, la razón principal por la cual cada uno de estos enfoques permite tener múltiples interfaces varía considerablemente:
En el enfoque de las "vistas", diferentes interfaces representan diferentes vistas de los clientes sobre una misma abstracción.
En el "modelo de roles" se necesitan para que un objeto tenga la habilidad de recibir y enviar diferentes mensajes en diferentes estados de evolución. Esta noción es casi totalmente ortogonal a la noción de "modo" o "vista".

Desde el punto de vista técnico, el enfoque dado por las fuerzas y los determinantes que se plantea en este trabajo , junto con el enfoque de los “modos”, parece ser el más pragmático y de bajo nivel entre todos los enfoques mencionados anteriormente porque ambos modelos limitan a que todos los determinantes de una fuerza definidos en una determinada clase, tengan una misma interface básica. En los otros enfoques, las interfaces de los objetos pueden llegar a contener un conjunto completamente distinto de operaciones.

Esto es una distinción muy importante, ya que de todos los enfoques anteriores, solamente el nuestro y el de utilización de “modos” pueden lograr obtener un polimorfismo a nivel de objetos de manera no trivial.

Sobre los Patterns de Diseño

Recordemos que John Vlisides en su trabajo denominado “Pattern Hatching: Subject-Oriented Design” [VLI/98], opinaba que con Subjetividad quizás se pueda llegar a obviar algunos patterns de diseño de hoy. Como hemos dicho en la sección de la Fuerza de Estado, consideramos que esto ocurre con el Pattern State. Resumimos las semejanzas y similitudes encontradas:

- El propósito de este Pattern y la Fuerza de Estado son equivalentes, es decir, ambos pueden resolver el mismo tipo de problema: que un objeto varíe su comportamiento de acuerdo con su estado interno.

- La Fuerza de Estado es aplicable en los mismos casos que el Pattern State.

- Con la Fuerza de Estado se mantienen las mismas ventajas, pero no se incrementa el número de clases como en el Pattern State:

  - Al utilizar el Pattern State se debe generar una clase por cada estado. Con la fuerza de estado no ocurre este problema ya que los diferentes estados son determinantes dentro de la fuerza y están dentro de la misma clase cuyo comportamiento se quiere variar.

  - Los estados se pueden compartir mediante el mecanismo habitual de “sharing”: la herencia. Las transiciones de estado pueden ser definidas de la forma en que se describe en la sección de “Las Fuerzas”, utilizando una lógica de fuerzas apropiada. Esto evita que las subclases de State tengan que conocerse entre sí y depender una de otra.

  - El comportamiento específico de cada estado está localizado en los métodos asociados a ese estado.

Retomando el ejemplo de la clase TCPConection, explicaremos en más nivel de detalle como la lógica de fuerzas implementará las transiciones de estado a nivel de método según el siguiente diagrama de transiciones:

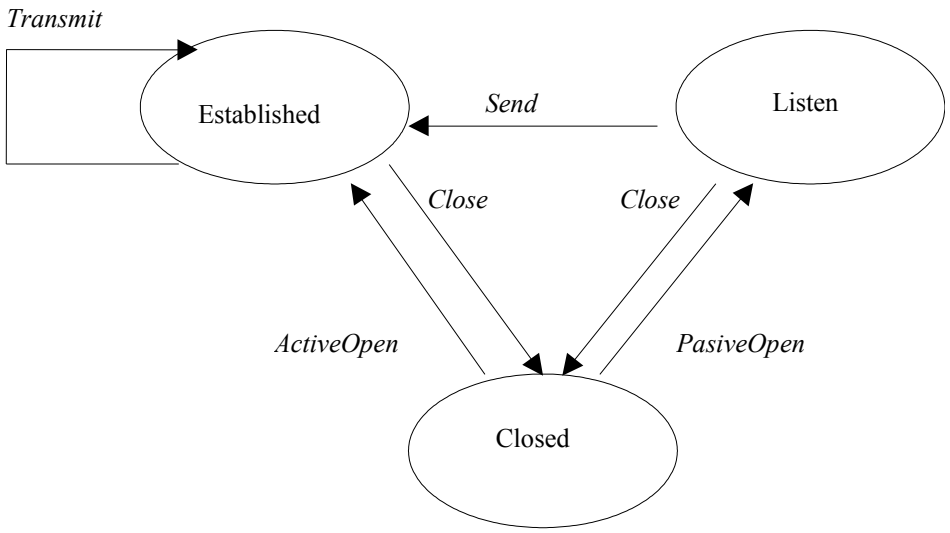


Figura 39: Autómata para las transacciones de Estado de TCPConnection

A nivel de cada método, la implementación de la lógica de fuerzas determinará el estado actual del objeto, luego de la ejecución del método. Esto resultará en una lógica de fuerza para cada método similar a las siguientes :

- Lógica de Fuerzas para ActiveOpen:
  anInfluenceForce force: 'State'.
  (State = 'Closed')
  ifTrue: [anInfluenceForce determinant:'Established']

- Lógica de Fuerzas para PasiveOpen:
  anInfluenceForce force: 'State'.
  (State = 'Closed')
  ifTrue: [anInfluenceForce determinant:'Listen']

- Lógica de Fuerzas para Close:
  anInfluenceForce force: 'State'.
  (State = 'Established')
  ifTrue: [anInfluenceForce determinant:'Closed']
  (State = 'Listen')
  ifTrue: [anInfluenceForce determinant:'Closed']

- Lógica de Fuerzas para Transmit:
  anInfluenceForce force: 'State'.
  (State = 'Established')
  ifTrue: [anInfluenceForce determinant:'Established']

- Lógica de Fuerzas para Send
  anInfluenceForce force: 'State'.
  (State = 'Listen')
  ifTrue: [anInfluenceForce determinant:'Established']

Los métodos para los cuales no se definió lógica de fuerza son aquellos en los cuales el comportamiento no varía según el estado interno, y no es necesario que sean subjetivos.

Nuestra conclusión es que con el uso de la Fuerza de Estado parecería que se pueden obtener resultados semejantes a los obtenidos con el uso del Pattern State, sin el agravante del aumento de número de clases.

Un caso similar ocurre con el Pattern Strategy. Este Pattern permite definir una familia de algoritmos, haciéndolos intercambiables y logrando que varíen en forma independiente de los clientes que lo utilicen. Si se considera que el comportamiento variante de los algoritmos pueden ser implementados cada uno por diferentes métodos, se podría utilizar la Fuerza del Emisor y lograr que estos algoritmos varíen de acuerdo al emisor del mensaje. Es decir, que según quién sea el emisor del mensaje, se cambie el comportamiento solicitado a al objeto en cuestión.

- El propósito del Pattern Strategy no coincide con el de la Fuerza del Emisor, aunque esta Fuerza puede utilizarse para encapsular un familia de algoritmos y hacerlos intercambiables.
- La Fuerza del Emisor permite ser aplicada en los siguientes casos en se utiliza el Pattern Strategy:
  - Se necesitan diferentes variantes de un algoritmo.
  - Una clase define muchos comportamientos y estos aparecen como sentencias condicionales múltiples en sus operaciones. (Las ramas condicionales se definen en los diferentes métodos subjetivos)
- Se pueden mantener las mismas ventajas usando la Fuerza del Emisor que con el Pattern Strategy pero no es necesario que los clientes conozcan las distintas estrategias ya que el objeto que contiene a las estrategias (o a la familia de algoritmos) decide como se comportará según el emisor del mensaje. Además, las diferentes estrategias pasan a ser diferentes implementaciones de un mensaje, con lo cual no se aumenta el número de objetos de una aplicación.

Siguiendo el ejemplo del Pattern Strategy que figura en el apéndice, se podría definir para la Fuerza del Emisor los siguientes determinantes:

- SimpleCompositor
- TeXCompositor
- ArrayCompositor

El método Compose puede definirse como un método subjetivo de la clase Composition , pero el cliente no necesitará definir cual es el compositor que se quiere utilizar. Se podría definir una colección de Emisores “Amigos” que se conocen que utilizarán una u otra implementación, como por ejemplo:

```
aComposition addKnowsSenders:[objeto11, objeto12,...objeto1n] toDeterminant: 'SimpleCompositor'.
aComposition addKnowsSenders:[objeto21, objeto22,...objeto2n] toDeterminant: 'TeXCompositor'.
aComposition addKnowsSenders:[objeto31, objeto32,...objeto3n] toDeterminant: 'ArrayCompositor'.
```

y en la lógica de fuerzas se puede definir la fuerza y el determinante que influyen en el comportamiento del objeto de manera similar a la siguiente:

```
anInfluenceForce force: 'Sender'.
anInfluenceForce determinant: (determinantInKnowsSender: sender)
```

Resultados Obtenidos

Durante el desarrollo de la tesis quedaron a la luz varias virtudes brindadas por la "Subjetividad" en un ambiente orientado a objetos. Ellas son:

- Permite modelar en forma más natural los objetos del mundo real, ayudando a disminuir un poco más el "gap" semántico, o sea: la distancia existente entre el mundo real y los modelos computacionales, y fortaleciendo uno de los objetivos principales de la Orientación a Objetos.

- Extiende el Paradigma de Orientación a Objetos, sin restringirlo en ningún sentido. No afecta ni modifica los conceptos básicos de la Orientación a Objetos. Veamos resumidamente qué sucede con los temas más importantes:

  - Encapsulamiento: la "Subjetividad" respeta este concepto, ya que los objetos encapsulan todos los mecanismos necesarios para realizar el despacho multi-métodos.

  - Ocultamiento de Información: la "Subjetividad" oculta a los clientes de clases subjetivas el hecho de que su comportamiento es subjetivo. Esto significa que su interface pública no revela la verdadera naturaleza del objeto receptor del mensaje, haciendo que la subjetividad sea transparente para los clientes de los objetos subjetivos.

  - Herencia: El mecanismo de herencia (desde el punto de vista conceptual) no se preocupa por el hecho de que el comportamiento heredado sea subjetivo o no. Todo el comportamiento subjetivo es heredado por todas las subclases de una clase dada.

  - Polimorfismo: Extiende este concepto logrando que los objetos sean polimórficos consigo mismos de manera no trivial. Si objetos de distintas clases pueden responder a un mismo mensaje de manera distinta, por qué no permitir que un mismo objeto pueda responder al mismo mensaje de manera diferente de acuerdo con el factor que lo influye, si de hecho es lo que sucede en la vida real?

- Permite modelar objetos con más "personalidad" e "inteligencia", ya que saben de qué manera responder de acuerdo con el factor que los influye.

- No fuerza a trabajar en un ambiente totalmente subjetivo, debido a que en el mundo real no todos los objetos varían su comportamiento ante el mismo mensaje de acuerdo con las circunstancias dadas. No todos los objetos deben ser subjetivos ni todos los mensajes de un mismo objeto tienen que ser subjetivos.

- Debido a que los mensajes que cada objeto puede recibir permanecen invariantes y solamente se modifica su implementación interna (método), evita la proliferación excesiva de métodos dentro de una misma clase.

- Disminuye también la proliferación excesiva de clases, dado que la "Subjetividad" reemplaza a la subclasificación en algunos casos, reduciendo de esta manera una explosión de la jerarquía de clases.

- Elimina muchas de las sentencias condicionales ("if" y "case").
  Sabemos que el hecho de contar con polimorfirsmo elimina muchas de las sentencias condicionales durante la programación. El agregado de Subjetividad, reduce otra cantidad de sentencias condicionales, ya que en el momento que un objeto debe responder a un mensaje, de acuerdo con el

factor que lo está influyendo, automáticamente ejecuta el método que corresponde. La estructura condicional, en este caso, estará presente pero en un solo lugar: dentro de la lógica de fuerzas. Y además, uno no necesita realizar la lectura de la lógica de fuerzas para entender, en un principio, el funcionamiento del objeto, agregando de esta forma mayor claridad a la programación.

- La "Subjetividad" definida con la fuerza "State" reemplazaría al pattern de diseño "State" y resolvería algunas aplicaciones del pattern de diseño "Strategy". [PRI/96] [GAM/95] [VLI/98]

Ventajas y Desventajas

A continuación analizaremos algunas ventajas y desventajas de la implementación propuesta en este trabajo.

Ventajas

- Es una herramienta que permite modelar en forma más natural el mundo real, utilizando la metodología de Orientación a Objetos.
- Se trata de un ambiente subjetivo mixto, donde el programador decide si utiliza o no la "Subjetividad" para una determinada clase.
- Permite incorporar el motor de subjetividad en cualquier ambiente standard sin necesidad de reconvertir código ya escrito.
- Permite extender las fuerzas que influyen a los objetos, ya que en la medida que comprendamos mejor cómo se comportan los objetos del mundo real, es muy posible que aparezcan nuevas fuerzas a modelar dentro del ambiente subjetivo.
- No existe precedencia de fuerzas definida previamente. El desarrollador es el que debe definirla.
- Respeta los conceptos básicos de la Orientación a Objetos.

Desventajas

- Tal como está planteada hoy en día la lógica de fuerzas, podría convertirse en un código demasiado extenso y complicado. Si bien una opción que se posee es la definición de lógica de fuerzas a nivel de método, esto puede no llegar a resultar práctico, con lo cual en un futuro habría que lograr mejorar este manejo.
- Los objetos del tipo "Force" no poseen demasiada inteligencia. Ya obtenida una implementación, habría que mejorar el modelo y quizás dando mayor inteligencia a estos objetos.

Futuros Trabajos

A partir de lo expuesto anteriormente, surgen algunos temas importantes para analizar y mejorar en un futuro sobre el tema de “Subjetividad”.

En primer lugar, sería conveniente intentar incorporar en forma adecuada los puntos expuestos en la sección “Desventajas”, de tal forma de enriquecer aún más el ambiente con el que contamos actualmente. No cabe duda que lo que poseemos en este momento es básicamente un prototipo que nos ha permitido estudiar el tema de Subjetividad. Alcanzado esto, sería conveniente realizar un rediseño que incluya los cambios propuestos.

Además deberían realizarse testeos más exhaustivos en cuanto a la funcionalidad del ambiente, con el fin de detectar su verdadero aporte en un proyecto de mayor envergadura que involucre “Subjetividad”. De esta forma se pondría realmente a prueba la madurez del ambiente propuesto.

Pero sin lugar a dudas, el tema de mayor importancia debe ser el relacionado con las **metodologías de diseño** en un ambiente subjetivo. Es evidente que junto con las herramientas deben emerger nuevas metodologías que ayuden a los desarrolladores a identificar y especificar el correcto comportamiento para cada una de las fuerzas. Debe ser capaz de establecer el modo en que las fuerzas influyen los objetos para poder especificar la lógica de fuerzas. Una idea inicial es comenzar con metodologías existentes y adaptarlas a nuestras necesidades.

Junto con las metodologías de diseño, habrá que considerar las notaciones asociadas a un ambiente subjetivo: definición de métodos subjetivos para cada fuerza/determinante, lógica de fuerzas, etc. Para ello quizás será necesario extender alguna herramienta que permita realizar esto.

Todo esto nos llevará a poder definir reglas de diseño para ambientes con subjetividad. Finalmente, deberíamos poder responder preguntas del estilo... “¿Cómo determinar bajo qué circunstancias debe utilizarse la Subjetividad?” “¿En qué impacta al diseño orientado a objetos el hecho que se utilice Subjetividad cuando en realidad no era necesaria?”.

A Modo de Final...

Ya terminado el actual proyecto, podemos decir que cumplimos el objetivo propuesto, contando ahora con un ambiente que permite la programación subjetiva.
Es difícil evaluar la madurez del ambiente desarrollado; lo más prudente antes de arriesgar un resultado final, sería utilizar este entorno para desarrollar distintos proyectos y obtener un mejor manejo sobre las funcionalidades necesitadas por los desarrolladores. Seguramente surgirán nuevos requerimientos y mejoras al ambiente obtenido. Pero lo más importante es que este trabajo nos dá el puntapie inicial necesario para continuar madurando el tema de la Subjetividad en un ambiente orientado a objetos.

Bibliografía

Decidimos clasificar la bibliografía en dos categorías:

- General: material de consulta con respecto a cuestiones generales del paradigma de orientación a objetos.
- Subjetividad: material que se refiere específicamente al tema de la tesis.

Bibliografía General

[AHO/90] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman
"Compiladores: Principios, técnicas y herramientas"
Addison-Wesley Iberoamericana.
1990.

[AMA/May93] Analía Amandi, Gustavo Rossi.
"Tecnología de Orientación a Objetos"
Revista "CompuMagazine".
Mayo 1993.

[AMA/Jun93] Analía Amandi, Gustavo Rossi.
"Tecnología de Orientación a Objetos"
Revista "CompuMagazine".
Junio 1993.

[BEC/93] Kent Beck.
"CRC: Finding objects the easy way"
Revista "Object Magazine".
Noviembre, Diciembre 1993.

[BEC/94] Kent Beck, Ralph Johnson.
"Patterns Generate Architectures"
Proceedings of the European Conference on Object-Oriented Programming (ECOOP'94).
Bologna, Italia. Julio 1994.

[BOO/94] G.Booch.
"Object-Oriented Analysis and Design with Applications" (Segunda Edición)
Addison-Wesley.
1994.

[BOO/99-R] G.Booch, I. Jacobson, J.Rumbaugh
"Unified Modeling Language – Reference Manual"
Addison-Wesley .
1998/1999.

[BOO/99-U] G.Booch, I. Jacobson, J.Rumbaugh
"Unified Modeling Language – User Guide"
Addison-Wesley .
1998/1999.

[COP/] James O. Coplien.
"Software Design Patterns: Common Questions and Answers"
Publicado en el libro "The Patterns Handbook: Techniques, Strategies, and Applications", (pag. 311-320), escrito por Linda Rising.
Cambridge University Press, New York.
Enero de 1998.

[DIE/87] Jim Diederich, Jack Milton.
"Experimental Prototyping in Smalltalk"
Revista "IEEE Software".
Mayo 1987.

[FER/89] Jacques Ferber.
"Computational Reflection in Class based Object Oriented Languages"
Proceedings of the Conference on Object-Oriented Programming Systems, Languages And Applications (OOPSLA'89)
New Orleans, LA, USA. Octubre 1989.

[GAM/95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.
"Design Patterns - Elements of Reusable Object-Oriented Software"
Addison Wesley.
1995.

[GLEN/] Glenn Krasner.
"The Smalltalk-80 Virtual Machine"
Revista "BYTE".
Agosto, 1981.

[GOL/83] Adele Goldberg, David Robson.
"Smalltalk-80: The Language and its Implementation"
Addison Wesley.
1983.

[GRA/89] Nicolas Graube.
"Metaclass Compatibility"
Proceedings of the Conference on Object-Oriented Programming Systems, Languages And Applications (OOPSLA'89)
New Orleans, LA , USA. Octubre 1989.

[HAL/87] Daniel C. Halbert, Patrick D. O'Brien.
"Using Types and Inheritance in Object-Oriented Programming"
Revista "IEEE Software".
Septiembre 1987.

[HAY/94] Wayne Haythorn.
"What is object-oriented design?"
Revista "Journal of Object Oriented Programming" (JOOP).
Marzo-Abril 1994.

[HEL/93] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.
"Design Patterns: Abstraction and Reuse of Object-Oriented Design"
Proceedings of the European Conference on Object-Oriented Programming (ECOOP'93).
Kaiserslautern, Alemania. Julio 1993.

[HEN/90] Brian Henderson-Sellers, Julian M. Edwards.
"The Object-Oriented Systems Life Cycle"
Communications of the ACM.
Vol.33, No.9.
Septiembre 1990.

[JAC/92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard.
"Object-Oriented Software Engineering - A Use Case Driven Approach"
Addison Wesley.
1992.

[JOH/88] Ralph Johnson, Brian Foote.
"Designing Reusable Classes"
Revista "Journal of Object-Oriented Programming" (JOOP).
Junio/Julio 1988.

[JOH/90] Rebecca Wirfs-Brock, Ralph E. Johnson.
"Surveying. Current Reasearch in Object-Oriented Design"
Communications of the ACM.
Vol.33, No. 9.
Septiembre 1990.

[JOH/92] Ralph Johnson.
"Documenting Frameworks using Patterns"
Proceedings of the Conference on Object-Oriented Programming Systems, Languages And Applications (OOPSLA'92)
Vancouver, B.C., Canadá. Octubre 1992.

[MEY/87] Bertrand Meyer.
"Reusability: The Case for Object-Oriented Design"
Revista "IEEE Software".
Marzo 1987.

[SHE/95] George Shepherd.
"The Timeless Way of Developing Software"
Revista "Software Development".
Mayo 1995.

[SIE/95-InfoNote 007] Linda Siener.
"VisualWorks Portability"
VisualWorks InfoNote 007.
Agosto 1995.

[SIE/95-InfoNote 010] Linda Siener.
"The Benefits of Object-Oriented Technology"
VisualWorks InfoNote 010.
Agosto 1995.

[STE/88] Lynn Andrea Stein, Henry Lieberman, David Ungar.
"A shared view of sharing: The Treaty of Orlando"
Brown University Technical Report.
Octubre, 1988.
(También en el Capítulo 3 de "Object Oriented Concepts, Applications and Databases", Won Kim, Fred Lochovsky, editors, ACM Press, 1988).

[UNG/87] David Ungar, Randall B. Smith
"Self: The power of simplicity"
Proceedings of the Conference on Object-Oriented Programming Systems, Languages And Applications (OOPSLA'87)
Orlando, FL, USA. Octubre 1987.

[WIL/95] Nancy Wilkinson.
"Using CRC Cards - An Informal Approach to Object Oriented Development"
SIGS Books.
1995.

[WIR/90] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener.
"Designing Object-Oriented Software"
Prentice Hall.
1990.

[WOO/97] Bobby Woolf
"Polymorphic Hierarchy: Subimplementro methods are the key"
Revista "The Smalltalk Report".
Enero, 1997.

Bibliografía sobre Subjetividad

IBM. "Subjective Object Programming" (SOP)
http://www.research.ibm.com/sop

[ALE/95] P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena.
"A Formal Model to Support Subject-Oriented Programming"
Object-Oriented Programming Systems Languages And Applications, Workshop sobre "Subjectivity in Object Oriented Systems" (OOPSLA'95).
Austin, TX, USA. Octubre 1995.

[BAR/] Daniel Bardou, Christophe Dony
"Split Objects: a Disciplined Use of Delegation within Objects"
Proceedings of the Conference on Object-Oriented Programming Systems, Languages And Applications (OOPSLA '96)
San Jose, CA, USA. Octubre 1996.

[FOO/95] Brian Foote.
"An Objective Look al Subjectivity"
Object-Oriented Programming Systems Languages And Applications, Workshop sobre "Subjectivity in Object Oriented Systems" (OOPSLA'95).
Austin, TX, USA. Octubre 1995.

[HAR/93] William Harrison, Harold Ossher.
"Subject-Oriented Programming (A Critique of Pure Objects)"
Proceedings of the Conference on Object-Oriented Programming Systems, Languages And Applications (OOPSLA'93)
Washington, DC, USA. Septiembre 1993.

[HAR/94] William Harrison, Harold Ossher, Randall B. Smith, David Ungar
"Subjectivity in Object-Oriented Systems – Workshop Summary"
Object-Oriented Programming Systems Languages And Applications, Workshop sobre "Subjectivity in Object Oriented Systems" (OOPSLA'94).
Portland, OR, USA. Octubre 1994.

[HAR/95] William Harrison, Harold Ossher, Hafedh Mili
"Subjectivity in Object-Oriented Systems – Workshop Summary"
Object-Oriented Programming Systems Languages And Applications, Workshop sobre "Subjectivity in Object Oriented Systems" (OOPSLA'95).
Austin, TX, USA. Octubre 1995.

[KRI/95] Bent Bruun Kristensen
"Subjectivity & Roles"
Object-Oriented Programming Systems Languages And Applications, Workshop sobre "Subjectivity in Object Oriented Systems" (OOPSLA'95).
Austin, TX, USA. Octubre 1995.

[LIE/86] Henry Lieberman
"Using Prototypical Objects to Implement Shared Behaviour in Object Oriented Systems"
Proceedings of the Conference on Object-Oriented Programming Systems, Languages And Applications (OOPSLA'86)
Portland , OR, USA. Noviembre 1986.

[MIL/96] Hafedth Mili, William Harrison, Harold Ossher
"Supporting Subject-Oriented Programming in Smalltalk"
Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS'96)
USA. Agosto 1996.

[OSS/94] Harold Ossher, William Harrison, Frank Budinsky, Ian Simmonds
"Subject-Oriented Programming:Supporting Descentralized Development of Objects"
Proceedings of the 7th. IBM Conference of Object- Oriented Technology.
Julio 1994.

[OSS/99] Harold Ossher, Peri Tarr.
"*Using Subject-Oriented Programming to Overcome Common Problems in Object-Oriented Software Development and Evolution*"
Proceedings of the 1999 International Conference on Software Engineering.
Los Angeles, CA, USA. 16-22 de Mayo de 1999.

[PER/90] Barbara Pernici.
"Objects with Roles"
Proceedings of the IEEE/ACM Conference on Office Information Systems.
Cambridge, MA, USA. Abril 1990.

[PRI/95] Máximo Prieto, Pablo Victory.
"Real World Object Behaviour"
Object-Oriented Programming Systems Languages And Applications, Workshop sobre "Subjectivity in Object Oriented Systems" (OOPSLA'95).
Austin, TX, USA. Octubre 1995.

[PRI/96] Máximo Prieto, Pablo Victory.
"Subjectivity: Towards True Polymorphism"
Object-Oriented Programming Systems Languages And Applications, Workshop sobre "Subjects and Viewpoints throughout the Life Cycle" (OOPSLA'96).
San Jose, CA , USA. Octubre 1996.

[PRI/97] Máximo Prieto, Pablo Victory.
"Subjective Object Behaviour"
Revista "Object Expert".
Vol.2/3. Marzo/Abril 1997.

[RIE/95] Dirk Riehle.
"Subjectivity in Object-Oriented Systems"
Object-Oriented Programming Systems Languages And Applications, Workshop sobre "Subjectivity in Object Oriented Systems" (OOPSLA'95).
Austin, TX, USA. Octubre 1995.

[TAI/93] Antero Taivalsaari.
"Object-oriented programming with modes"
Revista "Journal of Object Oriented Programming" (JOOP).
Junio 1993.

[VLI/98] John Vlissides
"Subject-Oriented Design"
Revista "C++ Report".
Febrero 1998.

[WER/95] Michael Werner. College of Computer Science.
"Why Use Subject-Oriented Programming in Databases"
Object-Oriented Programming Systems Languages And Applications, Workshop sobre "Subjectivity in Object Oriented Systems" (OOPSLA'95).
Austin, TX, USA. Octubre 1995.

[WIL/95] Jack C. Wileden, Alan Kaplan.
"Our SPIN on Subjectivity"
Object-Oriented Programming Systems Languages And Applications, Workshop sobre "Subjectivity in Object Oriented Systems" (OOPSLA'95).
Austin, TX, USA. Octubre 1995.

Apéndice I

Conceptos Básicos de la Orientación a Objetos

Los sistemas de computación tienen que representar problemas del mundo real, motivo por el cual diseñadores y programadores se enfrentan a problemas con diferentes grados de complejidad, entre los cuales podemos mencionar los siguientes:

- Modelar una aplicación puede requerir un gran esfuerzo. Las aplicaciones típicas son tan grandes y complejas que necesitan más de una persona para entenderla. Toma un largo tiempo desarrollar e implementar un software.

- Los cambios en el mundo real o las nuevas funciones del negocio que van surgiendo hacen que la aplicación deba sufrir modificaciones, teniendo que llegar en muchas ocasiones a un mantenimiento pesado. Una aplicación puede estar modificándose muchas veces a lo largo de su existencia, ir acumulando parches y pasando por diferentes versiones. Cuanto más se la arregle, más difícil será seguir arreglándola.

- Sería deseable que las nuevas aplicaciones pudieran ser construidas a partir de componentes ya desarrollados y probados; esto ayudaría a evitar la propagación de errores y el exceso de programación por similitud.

Las metodologías tradicionales no favorecen el tratamiento de estos problemas al dispersar en el sistema la información (datos) y el comportamiento (funcionalidad) de los componentes. De esta manera, cuando se debe modificar el comportamiento de algún componente o la representación de los datos habitualmente resulta difícil definir cual será el impacto que tendrá la modificación sobre el resto de los componentes del sistema.

Los conceptos de la orientación a objetos apuntan a solucionar los siguientes problemas:

- Necesidad de construir un software fácilmente **extensible** y **modificable**.
- Necesidad de disponer de los medios para obtener software **reusable**.
- Necesidad de disminuir el hueco semántico entre el mundo de problemas y el mundo de modelos.

Desde este punto de vista, una aplicación está compuesta por un conjunto de objetos que interactúan entre sí. Cada objeto encapsula un comportamiento determinado.

El rol de la Abstracción

La abstracción es una importante herramienta para tratar cuestiones complejas. Es esencial para entender el mundo real, en función de modelarlo en un software. Un buen ejemplo es un mapa, ya que debe ser significativamente más pequeño que su territorio e incluir solo la información seleccionada cuidadosamente.
La abstracción es un buen mecanismo para la descomposición del sistema en objetos. Estos objetos, a su vez nos permiten tener un mayor nivel de abstracción.

Los objetos

Como se dijo anteriormente, un objeto encapsula un cierto comportamiento. El comportamiento está caracterizado por los mensajes que el objeto puede recibir. A este conjunto de mensajes se lo denomina protocolo.

Mediante un objeto se puede modelar un objeto del mundo real, una estructura de datos, una transacción o lo que uno quiera. Cada objeto componente de una aplicación es responsable de una parte de la aplicación. Estas responsabilidades son llevadas a cabo por el objeto, mediante el comportamiento que tiene definido.

Encapsulamiento

El encapsulamiento permite tratar al objeto como una sola entidad en el sentido que todo el conocimiento de un objeto (que puede ser mucho o poco, según como uno quiera) y su comportamiento se mantienen dentro del objeto como si fuera una cápsula. De esta forma, se establece una barrera, manteniendo juntos tanto la información del objeto como el comportamiento que realiza. El encapsulamiento nos ayuda a conceptualizar y tratar con complejidad.

Ocultamiento de Información

Los objetos no se ven como un encapsulamiento transparente, sino como una caja negra. El resto de los objetos conocen el comportamiento que tiene un objeto (el protocolo al que responde un objeto). Pero cómo el objeto realiza este comportamiento y cuál es su estado interno forma parte de la representación privada del objeto.

La gran ventaja del ocultamiento de la información es que el objeto puede cambiar las estructuras de datos mediante las cuales mantiene su estado interno o la secuencia de envío de mensajes mediante el cual lleva a cabo determinado comportamiento sin modificar su interfaz, con lo cual no afecta al resto de los objetos de la aplicación. Es útil para aumentar el nivel de abstracción.

El encapsulado y ocultamiento de la información permiten que el código generado sea modificable y extensible.

Mensajes

Un objeto accede a otro enviándole un mensaje:

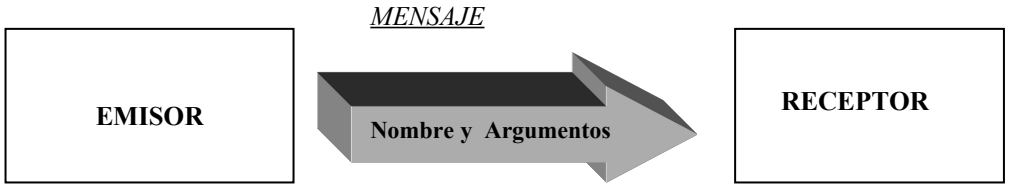


Figura 1 : Envío de un mensaje

Un *mensaje* consiste de un nombre de una operación y sus argumentos.

El objeto A que envía el mensaje (emisor) a otro objeto B (receptor), estará requiriendo que B ejecute cierta operación y (posiblemente) retorne alguna información. Cuando B recibe el mensaje, ejecuta la operación requerida mediante la ejecución de un *método*.

Un *método* es una secuencia de colaboraciones, o envíos de mensajes que ese objeto tiene con otros. No es un algoritmo. A lo sumo puede representar un algoritmo.

Los mensajes son habitualmente parte de la interfaz pública de un objeto, mientras que los métodos siempre forman parte de la representación privada. El conjunto de mensajes al que un objeto puede responder se conoce como *comportamiento del objeto*.

Clases e Instancias

Los objetos no son siempre uno diferente del otro. Una aplicación puede requerir muchas ventanas, archivos, colores o cuenta. Algunos objetos en una aplicación se comportarán diferentes unos de otros, y otros se comportarán de la misma manera.

Clases

Se dice que los objetos que tienen un mismo comportamiento pertenecen a una misma clase. Una clase es una especificación genérica para objetos similares. Se puede pensar en una clase como un template para un tipo específico de objeto. Permite construir una taxonomía de objetos en un nivel conceptual y abstracto. Las clases describen la estructura y el comportamiento común de un conjunto de objetos: de este modo uno puede crear objetos que se comporten de la manera indicada en la clase, cuando uno así lo requiera.

Los objetos que pertenecen a una misma clase poseen las mismas características, responden a los mismos mensajes y de la misma manera.

Instancias

Los objetos que se comportan en la manera especificada por una clase son llamados instancias de una clase. Todos los objetos son instancia de alguna clase. Una aplicación puede tener una o muchas instancias de una clase pero aunque se necesite una sola instancia (un SINGLETON en el contexto de los Patterns[GAM/95]), es necesaria la construcción de una clase. La creación de instancias de una clase es responsabilidad de la misma clase.

Las responsabilidades de una clase son representadas por los métodos de clase. En cambio, los métodos de instancia representan responsabilidades propias de objeto en particular.

Por ejemplo, uno podría identificar que todos los libros se comportan de la misma manera y especificar este comportamiento en la clase Libro. Así, cuando uno necesite crear un objeto libro, digamos 'Cien Años de Soledad', estará creando una instancia de la clase Libro mediante el envío de un mensaje "crear" a la clase Libro.

Otro ejemplo muy usado en Orientación a Objetos: una aplicación bancaria con una clase CajaDeAhorro. La responsabilidad de conocer e informar la cantidad máxima de extracciones que se pueden realizar en una caja de ahorro es propia de la clase y no de las instancias.

Polimorfismo

El acceso limitado de los objetos a interfaces definidas estrictamente, tales como el envío de mensajes permite otro uso de la abstracción, conocido como el polimorfismo: es la habilidad por la cual dos objetos pueden responder a un mismo mensaje de una manera semánticamente equivalente. Esto no significa que un objeto necesite saber a quién es enviado el mensaje, sino que muchos objetos responderán a aquel mensaje particular

Un buen ejemplo es la impresión de archivos con diferentes formatos. Cada clase de archivo contiene información para poder imprimirse correctamente. Cada una de estas clases implementará el *método print.* El objeto que solicite la impresión enviará simplemente el *mensaje print* a cada *objeto archivo* en cuestión, y cada uno de éstos ejecutará un *método print* distinto para imprimir su contenido, encapsulando en este método los detalles propios de la impresión.

Dos métodos son polimórficos si tienen el mismo nombre, el mismo tipo de parámetros, los mismo efectos colaterales, el mismo tipo de resultado y el mismo propósito, aunque no necesariamente los mismos detalles de implementación. Dos clases son polimórficas si todas sus instancias también lo son.

Herencia

Otro mecanismo de abstracción es la herencia: es la habilidad de una clase para definir el comportamiento y la estructura de datos de sus instancias como un superconjunto de la definición de otra clase u otras clases. Es decir, una clase es casi como otra clase, excepto por algunas cosas extras.
La herencia nos provee un mecanismo para la clasificación, con el cual podemos crear jerarquías de clases: se puede pensar a una nueva clase de objetos como un refinamiento de otra, para abstraer similitudes entre clases, diseñando y especificando solo las diferencias para las nuevas clases.

Si tomamos el caso de la clase Alumno, veremos que hereda el comportamiento de la clase Persona, ya que un alumno *es una* persona más su comportamiento propio.

La herencia permite establecer una relación entre dos clases (denominadas superclase y subclase). En toda relación de herencia hay una relación del tipo ES_UN.

Como consecuencia de la herencia se logra mejorar la reusabilidad del código, debido a que la clase puede heredar el comportamiento de otras clases. De esta forma la clase que usa el código que hereda, no necesita volver a implementar todo el código escrito en esos métodos.

Subclases y Superclases

Una subclase es una clase que hereda el comportamiento de otra clase, así como también la representación de datos. Por lo general agrega su propio comportamiento definiendo sus propios tipos de objetos. Por ejemplo, supongamos que tenemos una clase Array que nos permite manejar un vector indexado de elementos. Si necesitáramos manejar Strings de caracteres podríamos crear una subclase String, que herede el comportamiento de Array y que además permita realizar comparaciones alfabéticas.

Una superclase es la clase de la cual se hereda el comportamiento. En el ejemplo anterior, Array es la superclase de String.

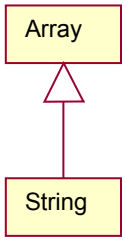


Figura 2 : Subclase y Superclase

Herencia simple y múltiple

Se dice que la herencia simple es aquella donde el comportamiento y la estructura interna se herede de a lo sumo una clase. En cambio, la herencia múltiple es aquella donde el comportamiento y la estructura interna se hereda de más de una superclase.

Volviendo al ejemplo anterior, podríamos contar con una clase Magnitude, cuyo comportamiento permitiría la definición de un orden entre los objetos. Luego String podría ser también subclase de Magnitude.

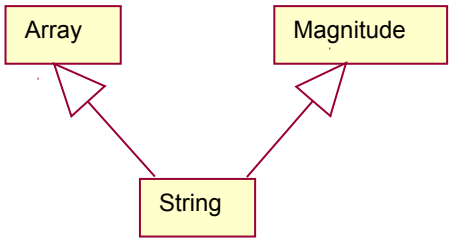


Figura 3 : Herencia Múltiple

Clases Abstractas y Concretas

No todas las clases necesitan crear instancias de sí mismas. La herencia puede ser un buen mecanismo para factorear el comportamiento común. En el ejemplo anterior, Magnitude existe para capturar en un lugar el comportamiento que implementa, no existirá ninguna instancia de esta clase. Se llaman clases abstractas a aquellas clases que no producirán instancias de sí mismas. Estas clases especifican su comportamiento pero no necesitan ser completamente implementadas. Pueden definir métodos que se redefinen en subclases. La implementación por defecto de ciertos comportamientos ayuda a prevenir errores.

Las clases concretas heredan el comportamiento de su superclase abstracta y agregan otras habilidades para sus propios propósitos. Estas clases completamente implementadas crean instancias de sí mismas para hacer su trabajo útil en un sistema.

Envío de Mensajes

Un objeto conoce a otro objeto cuando:

- Un objeto es parte de otro, ej. : Motor – Auto
- Un objeto se da a conocer a otro en forma permanente
- Un objeto se da a conocer a otro en forma temporaria.

De alguna manera, el objeto conocido recibe un determinado mensaje. La búsqueda del método (*method lookup)* a ejecutar se produce de la siguiente manera:

- Busca el mensaje en su clase. Si encuentra el mensaje, ejecuta el método
- Si no lo encuentra, lo busca en su superclase. Seguirá con esta búsqueda hasta encontrarlo o llegar a la superclase de nivel superior en la jerarquía.
- Si no encontró el mensaje, se producirá un error ya que el objeto no podrá responder al mensaje.

Framework

Un framework es un conjunto de clases cooperantes que permiten reusar un diseño de una aplicación específica, para resolver un problema de un dominio particular. Por ejemplo, se podrían utilizar un framework para construir compiladores en diferentes lenguajes de programación. A partir de las clases abstractas del framework se crean subclases específicas de la aplicación que se quiere “customizar”.

El framework define la arquitectura de la aplicación, enfatizando el reuso del diseño. Sin embargo, los frameworks pueden tener elementos de código (tales como clases o métodos totalmente implementados)

Patterns de Diseño

Brevemente, describiremos a continuación que son los Patterns de diseño y explicaremos dos de los patterns más relacionados con la Subjetividad, basándonos principalmente en el libro *"Design Patterns - Elements of Reusable Object-Oriented Software"* [GAM/95].

Un Pattern es una abstracción de un diseño que ataca a un problema general de diseño que se presenta en distintos dominios de aplicaciones. Cada Pattern intenta describir en forma abstracta a un problema de diseño y propone una solución al mismo. El diseñador debe conocer como y cuando aplicarlo. La solución consta de la descripción de un conjunto de clases y objetos que colaboran entre sí que se aplicarán en un contexto particular.

Formalmente, en [GAM/95] se define que los "Patterns de diseño son descripciones de clase y objetos comunicantes, customizados para resolver un problema de diseño general en un contexto particular."

El uso de Patterns permite hacer diseños reusables: el mismo diseño será aplicado en diferentes contexto.

Esencialmente, un Pattern consta de:

1. El *Nombre del Pattern:* compuesto por una o dos palabras que permite la identificación del Pattern por parte de sus diseñadores.
2. El P*roblema*: describe cuando aplicar el Pattern. Explica tanto el problema como su contexto. Pueden ser problemas de un diseño específico, pueden ser clases u objetos típicos de un diseño inflexible o puede ser una lista de condiciones que deben cumplirse antes de aplicar el Pattern.
3. La *Solución:* describe los elementos que hacen al diseño, sus relaciones, responsabilidades y colaboraciones. La solución es general, no es un diseño o implementación específica ya que el Pattern es como un template que puede utilizarse en distintas situaciones.
4. Las *Consecuencias:* son los resultados, los costos y beneficios de la aplicación del Pattern. Aunque las consecuencias no se invocan al describir las de diseño, son críticas para evaluar alternativas de diseño y entender los costos y beneficios de la aplicación del Pattern.

A continuación se describen en forma resumida dos Patterns: State y Strategy.

Pattern State

Propósito:

Permite que un objeto altere su comportamiento cuando su estado interno cambia. El objeto aparentará cambiar de clase.

Motivación:

Considérese una conexión de red TCP, mediante la clase TCPConnection. Un objeto de esta clase puede estar en diferente estado: Establecida, Escuchando, Cerrada. Cuando este objeto recibe pedidos de otros objetos responde de manera diferente según su estado interno al momento de recibir el mensaje. Por ejemplo, el efecto de un pedido de "Abrir Conexión" depende de si la conexión está en estado cerrada o establecida. El Pattern State describe como la TCPConnection puede mostrar un comportamiento diferente para cada estado.

La clave está en la clase abstracta denominada TCPState que representa los estados de las conexiones de red.

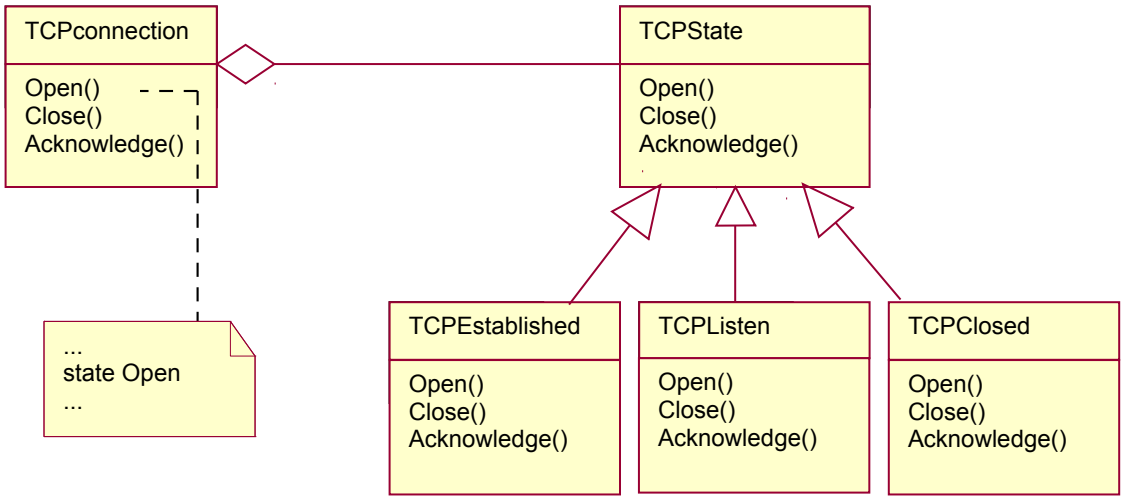


Figura 4 : TCPConnection modelada con el Pattern State

Aplicabilidad:

Use el Pattern State en los siguientes casos:

- El comportamiento de un objeto depende de su estado interno y debe cambiar su comportamiento en tiempo de ejecución, dependiendo de dicho estado.
- Los métodos tienen grandes sentencias condicionales que dependen del estado del objeto. Este estado se representa habitualmente por una o más constantes. Puede ocurrir que diferentes operaciones contengan la misma estructura condicional. El Pattern State pone cada rama del condicional en una clase separada. Le permite tratar el estado del objeto como un objeto en sí, pudiendo variar en forma independiente de los otros objetos.

Estructura:

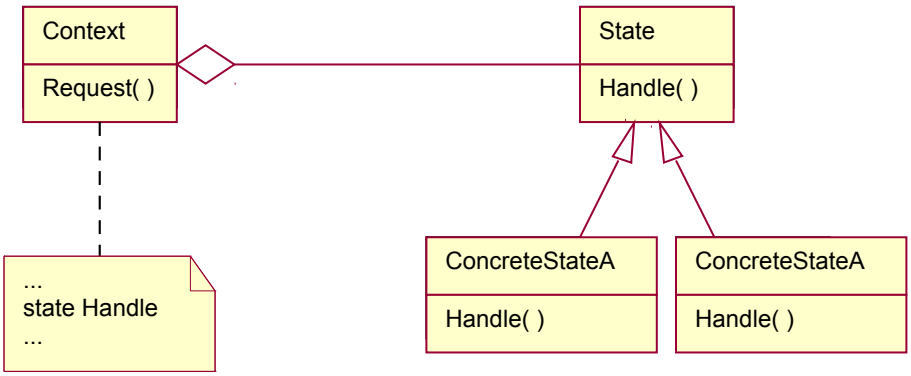


Figura 5 : Estructura del Pattern State

Participantes:

- **Context** (TCPConnection)
  - Define la interfaz de interés para los clientes
  - Mantiene una instancia de una subclase de ConcreteState que define el estado corriente.

- **State** (TCPState)
- Define una interfaz para encapsular el comportamiento asociado con el estado particular del Context

- **ConcreteState** (TCPEstablished, TCPClosed, TCPListen)
- Cada una de las subclases implementan el comportamiento asociado con el estado del Context.

Consecuencias:

Ventajas

- Localiza el comportamiento específico de estados y lo particiona para diferentes estados. Si bien se pueden generar una gran cantidad de subclases de State, de otra forma se generarían métodos monolíticos.
- Explicita las transacciones de estado.
- Los objetos estado pueden ser compartidos siempre y cuando no tengan variables de instancia.

Desventajas

- Incrementa el número de clases.
- Es menos compacto que una simple clase.

Pattern Strategy

Propósito:

Define una familia de algoritmos, encapsula a cada uno de ellos y los hace intercambiables. Permite a los algoritmos variar independientemente de los clientes que los usen

Motivación:

Muchos algoritmos existen para cortar un flujo de texto en líneas. La codificación de estos algoritmos dentro de las clases que los usan tiene algunos inconvenientes no deseados:

- Los clientes que necesitan las porciones de código se hacen cada vez más complejos si ellos incluyen la codificación. Esto hace a los clientes más grandes y más difíciles de mantener, especialmente si soportan la codificación de múltiples algoritmos.

- Diferentes algoritmos serán apropiados en diferentes veces. No se quiere soportar múltiples porciones de algoritmos si todas ellas no serán utilizadas

- Es difícil agregar nuevos algoritmos y variar los existentes cuando el código es una parte integral del cliente.

Estos inconvenientes se pueden evitar definiendo clases que encapsulan el código de los diferentes algoritmos. Un algoritmo encapsulado de esta forma se llama una **estrategia (strategy).**

La siguiente figura, muestra un ejemplo de una clase- Composition que es responsable de mantener y actualizar los corter de líneas de texto en una ventana de texto (*text viewer)* . Las estrategias para los cortes de líneas de texto se implementan en las subclases de la clase abstracta Compositor , en lugar de estar en la clase Composition.

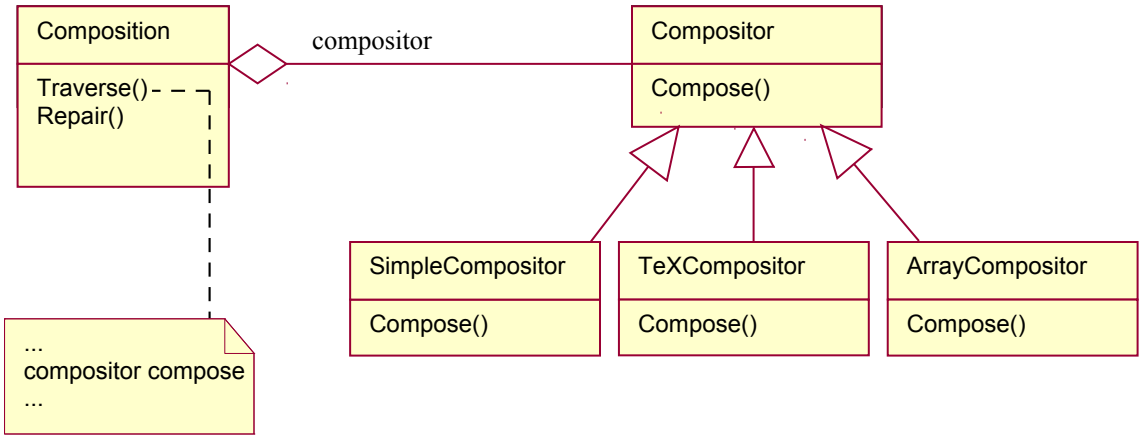


Figura 6 : Compositor para líneas de texto

Este diseño permite que las subclases de Compositor implementen diferentes estrategias:

- **SimpleCompositor** implementa una estrategia simple que determina los cortes de líneas de a uno a la vez.
- **TeXCompositor** implementa el algoritmo de T_EX para encontrar los cortes de líneas. La estrategia trata de optimizarlos globalmente, es decir, de a un parágrafo a la vez.
- **ArrayCompositor** implementa una estrategia que selecciona los cortes tal que cada fila tiene un número fijo de items. Es útil para cortar una colleción de íconos dentro de las filas.

Una Composición conoce a algún objeto Compositor, a quién le pasará la responsabilidad de reformatear su texto. El cliente de la Composición especifíca que Compositor debería ser usado, instalando el Compositor deseado dentro de la clase Composition.

Aplicabilidad:

El Pattern Strategy se usa cuando:

- Muchas clases relacionadas difieren solo en su comportamiento.
- Se necesitan diferentes variantes de un algoritmo.
- Un algoritmo usa datos que los clientes no deberían conocer.

- Una clase define muchos comportamientos y estos aparecen como sentencias condicionales múltiples en sus operaciones. En lugar de usar muchos condicionales, se mueven las ramas condicionales a su propia clase Strategy.

Estructura:

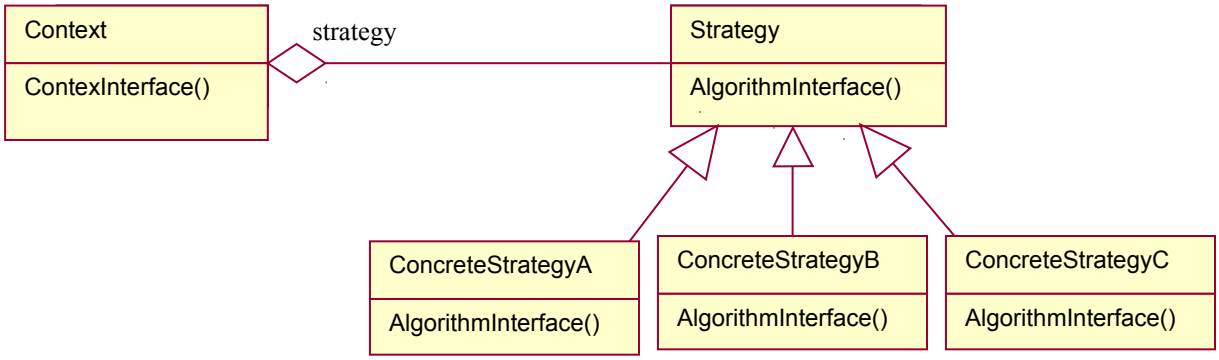


Figura 7 : Compositor para líneas de texto

Participantes:

- **Strategy** (Compositor)
  - Declara una interfaz común a todos los algoritmos soportados. Context la usa para llamar al algoritmo implementado por una ConcreteStrategy.
- **ConcreteStrategy** (SimpleCompositor, ArrayCompositor, TexCompositor)
  - Implementa el algoritmo usando la interfaz de Strategy.
- **Context** (Composition)
  - Es configurada con un objeto ConcreteStrategy
  - Conoce a un objeto Strategy
  - Puede definir una interfaz que le permita a Strategy acceder a sus datos.

Consecuencias:

Ventajas

- Permite manipular familias de algoritmos relacionados, haciendo uso de la herencia para factorizar la funcionalidad de los algoritmos
- Otorga una alternativa para la subclasificación: se podrían hacer subclases de Context directamente para manejar los distintos comportamientos, pero esto mezclaría la implementación de Context con el algoritmo y haría el código más difícil de entender, mantener, extender y variar dinámicamente.
- El uso de Strategies permite eliminar sentencias condicionales.
- El cliente puede elegir entre diferentes implementaciones para un mismo comportamiento, basado en la consideración de cuestiones como tiempo y espacio.

Desventajas

- Los clientes deben conocer las diferentes estrategias, entendiendo sus diferencias para poder seleccionarlas correctamente. Este Pattern solo debe usarse si la variación del comportamiento es lo suficientemente relevante para el cliente.
- Puede ocurrir una sobrecarga de comunicación entre Strategy y Context. Además, Context podrís crear e inicializar parámetros que algunas ConcreteStrategies no utilicen. Es necesario un acoplamiento estrecho entre Context y Strategy.

Apéndice II

Nociones de UML utilizadas en la Tesis

El UML, *Lenguaje Unificado de Modelado,* utilizado para modelar distintas situaciones durante el desarrollo de un proyecto. En este apéndice solo trataremos aquellos componentes que hemos utilizado durante el desarrollo de la Tesis.

Diagrama de Clases

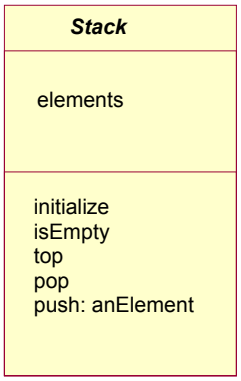
Un diagrama de clases se utiliza para describir visualmente diversos aspectos del sistema. Ayuda a representar el dominio del problema y también del sistema. Contiene clases, paquetes lógicos, objetos, métodos, paquetes de componentes, módulos, procesadores, dispositivos y las relaciones entre ellos. Cada uno de estos componentes posee propiedades que los identifica y caracteriza.

Durante la etapa de análisis, muestra los roles y responsabilidades comunes de las entidades que proveen el comportamiento del sistema. Durante el diseño, captura la estructura de las clases que forman la arquitectura del sistema.

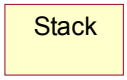
Clase

Una clase captura la estructura y el comportamiento en común de un conjunto de objetos. Una clase es una abstracción de un ítem del mundo real. Cuando estos ítems existen en el mundo real, son instancias de la clase, y referenciados como objetos.

Un ícono de clase es dibujado como una caja con tres divisiones. En la parte superior figura el nombre de la clase, la lista de variables de instancia en la parte media y una lista de mensajes en la parte inferior.

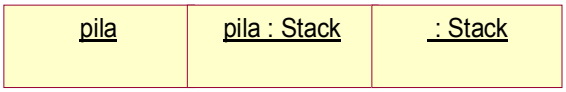


Las secciones de variables de instancia y de métodos pueden ser suprimidas para reducir los detalles a ser mostrados. Una sección vacía significa que no hay elementos en dicha sección.



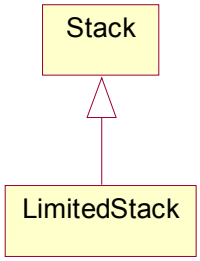
Objeto

Un objeto posee estado, comportamiento e identidad. La estructura y comportamiento de objetos similares son definidos en sus clases en común.
Cada objeto en un diagrama indica alguna instancia de una clase. El icono de un objeto es similar al de la clase, con la diferencia que el nombre esta subrayado. En este diagrama se puede indicar solo el nombre del objeto, o bien el nombre del mismo seguido del carácter dos puntos (:) y el nombre de la clase, o bien, si dicho objeto no tiene nombre se colocará el caracter dos puntos (:) seguido del nombre de la clase.



Relación de Herencia

Una relación de herencia entre dos clases muestra que la subclase comparte la estructura y el comportamiento definido en la superclase. Se utiliza para mostrar la relación "es un" entre dos clases.
Gráficamente se muestra como una línea sólida con una punta de flecha apuntando a la superclase.



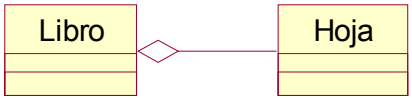
Relación de Asociación

Una relación de asociación representa una conexión semántica entre dos clases. Las asociaciones son bi-direccionales, son las más generales de todas las relaciones y semánticamente las más débiles.
Gráficamente una asociación se muestra como una línea entre dos clases.

Para facilitar la lectura del gráfico, se le puede agregar una flecha a la línea.

Relación de Agregación

Una relación de agregación muestra la relación de un todo y una parte entre dos clases. Se usa para mostrar que el objeto agregado esta construido físicamente desde otro objeto, o que contiene lógicamente otro objeto. La clase del lado del proveedor (supplier) es la parte cuyas instancias están contenidas o manejadas por el objeto cliente.
Gráficamente una agregación se muestra como una línea sólida con un rombo en el final, apuntando hacia la clase cliente.



En el ejemplo, el libro es el cliente, y la hoja el proveedor.

Notas aclaratorias

Una nota captura todos los asuntos y decisiones aplicados durante el análisis y diseño. Las notas pueden contener cualquier información, incluyendo texto, fragmentos de código, o referencias hacia otros documentos.
Gráficamente una nota se muestra como una caja con el ángulo superior derecho doblado.

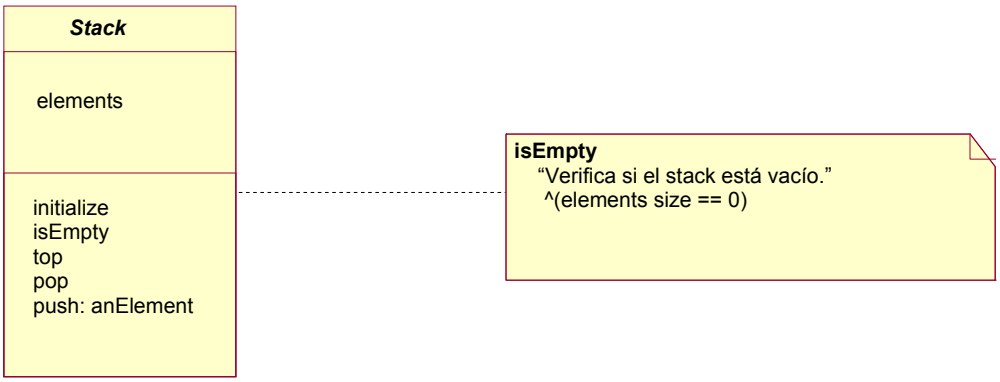
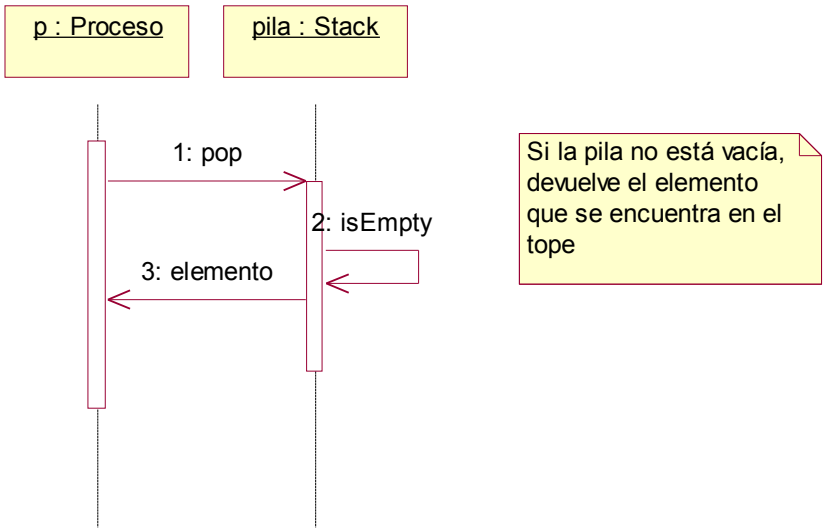


Diagrama de Secuencia

Un diagrama de secuencia traza la ejecución de una interacción entre objetos en el tiempo.
Contiene objetos, con una línea vertical que sale de cada uno, la que representa el paso del tiempo; rectángulos sobre las líneas, que representan el tiempo relativo en el que el foco de control esta en el objeto y flechas entre las líneas verticales que representan los mensajes que los objetos se envían y las respuestas devueltas. También pueden haber notas aclaratorias.



Apéndice III

Nociones Básicas sobre el Lenguaje Smalltalk

Smalltalk es un lenguaje de programación que permite elaborar sistemas compuestos por objetos, los cuales interactúan entre sí mediante el envío de mensajes. Está basado en clases: los objetos existen como instancias de las clases. Estas clases contienen la definición de los mensajes que los objetos pueden recibir, y la implementación de los métodos asociados a cada mensaje.

La sintaxis para el envío de un mensaje simplemente es: **objeto mensaje**. Por ejemplo, si **pila** es un objeto de la clase LimitedStack y se le quiera solicitar un pop, la sintaxis será:

pila pop

Esto provocará que se seleccione el método pop de la clase LimitedStack y se ejecute. Un método describe como un objeto llevará a cabo el comportamiento asociado al mensaje recibido, y está compuesto por una secuencia de colaboraciones, es decir, envíos de mensajes a otros objetos.

Expresiones Sintácticas

Las expresiones sintácticas que permiten describir objetos y mensajes en Smalltalk son:

- Literales: describen objetos constantes tales como números y cadenas de caracteres (strings):
    1. Números: son objetos que representa valores numéricos y responden a mensajes que computan resultados matemáticos: 3, 4.78 , -1000, etc.
    2. Caracteres: son objetos que representan símbolos individuales de un alfabeto. Se representan con el signo pesos ($) seguido del carácter: $A, $-, $1.
    3. Cadenas de Caracteres (Strings): representan secuencias de caracteres. Responden a mensajes para acceder a caracteres individuales o manipular subcadenas y comparar cadenas. Se representan como secuencias de caracteres delimitadas con una comilla simple: ‘hola’, ‘Smalltalk Strings’
    4. Símbolos: objetos que representan strings usadas por el sistema. Se representan con el símbolo numeral (#) seguido de la string: #pop, #new.
    5. Array: un array es un objeto con una estructura de datos que permite que sus contenidos sean referenciados con un índice entero de uno a un número que se corresponde con el tamaño del array. Se representa con una secuencia de otros literales, entre paréntesis y precedida por el signo numeral: #(1 2 3), (‘hola’ ‘uno’ ‘dos’).

- Nombres de variables: describe una variable a la cual se puede acceder. Una variable contiene algún objetos, su nombre es una expresión que permite referirse a dicho objeto. Tal nombre es un simple identificador: una secuencia de números y dígitos que comienzan con una letra. Ejemplo: aStack, index, CajaDeAhorro.

- Expresiones de mensajes: describen mensajes que se envían a receptores. El valor de la expresión es determinado por el método que se invoca al recibir el mensaje.

- Expresiones de bloques: describen objetos que representan actividades diferidas. Se usan habitualmente para implementar estructuras de control.

Variables

Como dijimos anteriormente, cada variable contiene algún objeto al cual se puede hacer referencia mediante el nombre de la variable. Debido a que Smalltak es un lenguaje no tipado, una variable puede referenciar en diferentes momento a diferentes objetos y de distintas clases.

Las variables se pueden distinguir en *privadas* o *compartidas.* Las variables privadas son accedidas solamente por un simple objeto, mientras que las variables compartidas son aquellas que pueden ser accedidas por varios objeto. Las privadas tienen nombres que empiezan con minúsculas y los nombres de las globales deben empezar con mayúsculas.

Hay dos tipos de variables privadas: las variables de instancia y las variables temporarias. Las primeras existen dentro del tiempo de vida de un objeto y representan el estado interno del objeto. Se especifican en el momento en que se crea la clase. En el siguiente ejemplo, se crea la clase LimitedStack, con la variable de instancia 'maxElementsQuantity'

```
Stack subclass: #LimitedStack
instanceVariableName : 'maxElementsQuantity'
```

Las variables temporarias son creadas para una actividad específica (típicamente dentro de un método) y están disponibles dentro de la duración de esa actividad. Por ejemplo, dentro del método **pop** se crea la variable temporaria **element**, para referenciar al objeto que se extrae de la pila. La sintaxis que se utiliza es el nombre de la variable entre dos símbolos pipes

```
| element |
elements at: (self size).
```

Otro caso de variables temporarias son los argumentos de los métodos y las variables temporales de bloques, que serán explicadas más adelante.

Las variables compartidas se pueden distinguir en tres tipos: las de clase, las globales y las pool. Las variables de clase son compartidas por todas las instancias de una sola clase. Las variables globales son compartidas por todos los objetos. Las variables pool son compartidas por instancias de un subconjunto de clases, por ejemplo los Pool Dictionary que pueden contener constantes de colores.

Asignación

Para asignar el valor a una variable se utiliza el símbolo **:=** . La siguiente expresión asigna a la variable **pila** un objeto de la clase LimitedStack.

```
pila:= LimitedStack new.
```

El método new está definido para la clase LimitedStack (método de clase), y permite la creación de instancias de la clase LimitedStack.

Pseudo-Variables

Un nombre de una pseudo-variable es un identificador que referencia a un objeto. Es similar a un nombre de variable. La diferencia es que el valor de una pseudo-variable no puede cambiar con una expresión de asignación. Algunas pseudo-variables tienen valor constante, entre ellas podemos citar las siguientes:

- **nil** : es una instancia de la clase UndefinedObject. Cuando una variable se crea, se le asigna el objeto **nil,** referenciado por esta pseudo-variable, hasta tanto se la inicialice con otro objeto.

- **true** : es una instancia de la clase True, representa una afirmación lógica. Es usado para responder en forma afirmativa a un mensaje que realiza una pregunta del tipo si-no.

- **false** : es una instancia de la clase False, representa una negación lógica. El uso es similar al del true pero para responder en forma negativa.

Self y Super

Otras pseudo-variables tienen diferentes valores, dependiendo de donde son invocadas. Dentro de un método la pseudo-variable **self** se refiere al objeto receptor del mensaje en sí mismo. Permite que dentro de un método, un objeto pueda enviarse mensajes a sí mismo. También facilita la elaboración de métodos recursivos.

La pseudo-variable **super** representa al receptor del mensaje, como **self**. Sin embargo, cuando un mensaje es enviado a **super**, la búsqueda para el método no comienza en la clase del receptor sino en la superclase de la clase que contiene el método. El uso de **super** permite que un método acceda a los métodos definidos en la superclase, sin tener que re-escribirlos.

Mensajes

Como dijimos anteriormente, los objetos interactúan entre sí mediante el envío de mensajes. Un mensaje es una expresión que describe un receptor, un selector y posiblemente algunos argumentos. El selector es el nombre del mensaje.
En el siguiente ejemplo, **pila** es el receptor del mensaje y **pop** es el selector del mensaje.

pila pop

En este otro ejemplo, **push:** es el selector del mensaje y el objeto referenciado por la variable **anElement** es el argumento del mensaje.

pila push: anElement

Como se explicó en la sección de asignación de variables, el valor retornado por un mensaje puede ser asignado a una variable. Por ejemplo, el elemento retornado por el mensaje **pop** podría ser asignado a la variable **myElement**:

MyElement: pila pop

Mensajes Unarios

Son mensajes sin argumentos. Es el caso del mensaje **pop**. Otros ejemplos:

'Hello World' size

20 factorial

Mensajes Binarios

Son mensajes que se utilizan principalmente para mensajes aritméticos. Tienen un único argumento que puede ser cualquier expresión válida. El selector está compuesto por uno o más por uno o dos caracteres alfanuméricos. Ejemplos:

2 + 5

precio * cantidad

precio + 20

minimo >= total

Mensajes de Palabra Clave

Son mensajes con uno o más argumentos. El selector está compuesto por una o más palabras claves, seguidas de un punto y coma (;). Los argumentos pueden ser cualquier expresión válida. Ejemplos:

pila push: anElement

enLetras at: 3 put: 'uno'

Precedencia

Los diferentes tipos de mensajes son evaluados de izquierda a derecha. Los unarios tienen precedencia sobre los binarios, y éstos la tienen sobre los de palabra clave. El orden de evaluación puede ser cambiado mediante el uso de paréntesis. Por ejemplo:

3 * alpha factorial.

(3 * alpha) factorial.

En el primer caso, se calcula 3 veces el factorial de alpha. En el segundo caso se calcula el factorial de (3 *alpha)

Métodos

Un método está formado por una secuencia de envíos de mensajes a otros objetos. Se compone de un patrón del mensaje y una secuencia de expresiones separadas por el carácter punto ".". Dentro del método se pueden declarar variables temporarias. Los comentarios pueden ser expresados entre comillas dobles ". Por ejemplo, los métodos **pop** y **push** son los siguientes:

```
pop
  "Retorna el tope del stack, eliminandolo del mismo."

   | element |
   (self isEmpty)
      ifTrue: ["El stack está vacío".
               ^nil ]
      ifFalse:[ element:= elements removeAtIndex:
                          (element size).
                ^element.].

push: anElement
  "Agrega anElement al stack."
   elements addLast: anElement.
```

Los patrones de mensaje son ***pop*** y ***push: anElement*** respectivamente. Contienen el selector del mensaje y los nombres de los argumentos (si corresponde). El carácter ^ indica el valor de retorno del método.

Bloques

Son secuencias de expresiones separadas por el carácter punto "." y encerradas entre corchetes "[ ]". Estas expresiones no son ejecutadas inmediatamente, sino que en el momento en que son requeridas. Por ejemplo,

```
pila := [ | p | p:= LimitedStack new.
                    p push: 1.
                    p push: 2
          ]
```

La ejecución de la expresión anterior no implica que la pila tenga los elementos 1 y 2, sino simplemente que pila contiene al bloque en cuestión.

La secuencia de acciones de un bloque se ejecutarán cuando el bloque reciba el mensaje **value**. Por ejemplo:

```
pila value.
```

Esto provocará que se cree un LimitedStack, al cual se le agreguen los elementos 1 y 2. Una vez finalizada la ejecución del bloque, el LimitedStack dejará de existir, ya que es una variable temporaria cuyo tiempo de vida está limitado a la ejecución del bloque.

Un bloque puede tener argumentos. En el siguiente ejemplo, **a** es un argumento del bloque, que al evaluarse se instanciará con el valor de **miArchivo**. La evaluación del bloque provocará que se envíe el mensaje **print** al objeto referenciado por la variable **miArchivo**.

```
x := [ : a | a print]

x value: miArchivo
```

Estructuras de Control

Los bloques permiten manejar estructuras de control, entre ellas se destacan las estructuras condicionales. Por ejemplo las sentencias condicionales del tipo IfTrue, IfFalse son mensajes enviados a los objetos *Booleanos true* o *false* (de las clases *True o False respectivamente*). Por ejemplo,

```
(cantidad <= 20)
        ifTrue: [ total = precioUnitario * cantidad ]
        ifFalse: [ total = precioUnitario * cantidad *0.90]
```

Dependiendo del objeto referenciado por la variable cantidad, la condición cantidad <=20 retornará el objeto *true* o *false*. Si retorna *true* se enviará el mensaje ifTrue:IfFalse al objeto *true*, lo que ejecutará el método implementado en la clase *True*, que tiene la siguiente codificación:

```
ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
        ^trueAlternativeBlock value
```

En caso contrario, si la expresión retorna *false* se enviará el mensaje al objeto *false*, lo que ejecutará el método implementado en la clase *False*, el cual se define como:

```
ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
        ^falseAlternativeBlock value
```

De manera similar se pueden implementar estructuras de repetición. Por ejemplo, enviando a un bloque un mensaje **whileTrue:** y otro bloque como argumento del mensaje. El bloque receptor se envía a sí mismo el mensaje **value** y si se devuelve un objeto *true* evalúa el bloque pasado como argumento. Luego se envía nuevamente el mensaje **value**. Cuando este mensaje retorne un objeto *false* parará las repeticione y terminará la ejecución del método **whiletrue**. El siguiente ejemplo decrementa de a 10 el valor del objeto referenciado por la variable cantidad, hasta tanto sea menor que el valor de tope.

```
[ cantidad < tope ] whiletrue:[ cantidad := cantidad  –10]
```

Es interesante observar las estructuras de control en los casos de objetos que representan estructuras de datos, tal como el mensaje **do:**, El objeto que recibe este mensaje evalúa el bloque una vez para cada elemento de su estructura de datos. El siguiente ejemplo calcula la suma de los cuadrados de los cinco primeros números primos:

```
sum:=0.
#(2 3 5 7 11) do: [ :prime | sum := sum + (prime * prime) ]
```

El mensaje **collect:** crea una colección de los valores retornados por el bloque. El resultado del siguiente ejemplo es un array de los cuadrados de los cuadrados de los cinco primeros números primos.

```
#(2 3 5 7 11) collect: [ :prime | prime * prime ]
```

Apéndice IV

Pasos a seguir para generar el ambiente subjetivo dentro de VisualWorks 3.0

Pre-Requisitos:

1. Tener el VisualWorks Non-Commercial, Release 3.0 instalado y funcionando correctamente.
2. Contar con los archivos correspondientes a las clases que se necesitan para extender el ambiente tradicional a un ambiente subjetivo. Son las clases desarrolladas en la tesis. Los archivos son los siguientes:

NOMBRE DE LA CLASE	NOMBRE DEL ARCHIVO (formato largo)
Force	Force.st
ForceAdministrator	ForceAdministrator.st
ForceLogicDefinition	ForceLogicDefinition.st
ExecutionContext	ExecutionContext.st
SubjectiveMethod	SubjectiveMethod.st
InfluenceForce	InfluenceForce.st
SubjectivityBehavior	SubjectivityBehavior.st
NullCompiledMethod	NullCompiledMethod.st
SubjectivityCompiledMethod	SubjectivityCompiledMethod.st
SubjectivityBrowser	SubjectivityBrowser.st
SubjectivityDebugger	SubjectivityDebugger.st
SubjectivityVisualLauncher	SubjectivityVisualLauncher.st
NotifierView	NotifierView.st

Instalación:

A. Abrir el VisualWorks.

B. Utilizando el menú, “Tools”\”File List”, proceder a importar (“File-In”), las siguientes clases, en el orden que se indica:
   1. Force
   2. InfluenceForce
   3. ForceAdministrator
   4. ForceLogicDefinition
   5. ExecutionContext
   6. SubjectiveMethod
   7. SubjectivityBehavior
   8. NullCompiledMethod
   9. SubjectivityCompiledMethod
   10. SubjectivityBrowser
   11. SubjectivityDebugger
   12. SubjectivityVisualLauncher
   13. NotifierView

C. Definir la **variable global** “SubjectivitySmalltalk” realizando un “Do It” al siguiente texto:
   SubjectivitySmalltalk := SubjectivityBehavior new initialize.

D. Abrir el Visual Launcher Subjetivo, realizando un “Do It” al siguiente texto:
   SubjectivityVisualLauncher open.

Esto abrirá una nueva ventana cuyo título será “Subjective VisualWorks Non-Commercial”.

E. Cerrar el Visual Launcher Normal abierto al momento de ingresar al VisualWorks; es la ventana denominada “VisualWorks Non-Commercial”; responder afirmativamente cuando se nos solicite confirmación.

F. Grabar la imagen actual y salir del VisualWorks, realizando “File”\”Exit VisualWorks”\”Save then Exit”. Aquí es conveniente cambiar el nombre del archivo de imagen. Por ejemplo, si el archivo actual se denomina “visualnc”, se lo puede cambiar por “subjvisualnc”, y dar “OK”.

G. Si en el punto anterior no se renombró el archivo, continuar con el siguiente paso. Sino, se debe proceder a asociar la nueva imagen al archivo de ejecución del VisualWorks.
(Windows’95 / NT): Si se posee un ícono de acceso rápido en el área de trabajo, editar las propiedades, y donde se encuentra la asociación, escribir el nombre del archivo de imagen que se creó en el punto anterior.

H. Dar de alta las fuerzas existentes en el ambiente subjetivo, utilizando el “Administrador de Fuerzas” generado para tal fin. Para ello, hacer: “Browser”\”Admin Forces”. Esto abrirá una ventana que permitirá ingresar las fuerzas al ambiente. Hasta este momento, valdrá la pena ingresar las siguientes fuerzas:

   - State
   - Sender
   - Content

   En cualquier momento se podrá recurrir a esta opción para agregar una nueva fuerza al ambiente.

I. Grabar la imagen realizando “File”\”Save As” y respetando el nombre del archivo de imagen.

¡ FIN DE LA INSTALACION DEL AMBIENTE SUBJETIVO !

Apéndice V

El código de las clases que forman el ambiente subjetivo para VisualWorks 3.0

CLASE ExecutionContext

'From VisualWorks® Non-Commercial, Release 3.0 of May 8, 1998 on January 9, 2000 at 8:18:04 am'!

Object subclass: #ExecutionContext
instanceVariableNames: 'subjectClass subject sender arguments selector '
classVariableNames: ''
poolDictionaries: ''
category: 'SubjectiveEnvironment'!

ExecutionContext comment:
'Esta clase contiene el contexto de ejecucion necesario para poder manejar comportamiento subjetivo. Colabora principalmente con la logica de Fuerzas (ForceLogic) para poder manejar la subjetividad. Sus instancias son creadas por el SubjectivityCompiledMethod, inicializando los valores de sus variables de instancia con la informacion del contexto de ejecucion al momento de recibir el mensaje

subjectClass	<Class>	Clase a la cual pertenece el sujeto (receptor del mensaje subjetivo)
subject	<>	Puede ser cualquier clase, es el objeto receptor del mensaje
sender	<>	Puede ser cualquier clase, es el objeto emisor del mensaje
arguments	<Array of: Objects>	Son los parametros del mensaje
selector	<Symbol>	Es el nombre del mensaje (selector del metodo)'!

!ExecutionContext methodsFor: 'accesing'!

arguments
"Retorna un array con los parametros actuales con que se invoca al mensaje."

^arguments!

arguments: anArrayOfArguments
"Setea los parametros actuales con que se invoco al mensaje con el valor dado en anArrayOfArguments. Se asume que el tamano del Array coincide con la cantidad de parametros especificados (o sea con la cardinalidad del mensaje)."

arguments := anArrayOfArguments!

selector
"Devuelve el selector del mensaje invocado"

^selector!

selector: aSelector
"Setea el selector del mensaje que se esta invocando"

selector := aSelector.!

sender
"Retorna el objeto emisor del mensaje."

^sender!

sender: aSender
"Setea el objeto emisor del mensaje."

sender := aSender!

subject
"Retorna el objeto receptor del mensaje."

^subject!

subject: aSubject
"Setea el objeto receptor del mensaje."

subject:= aSubject!

subjectClass
"Retorna el nombre de la clase subjetiva a la que pertenece el objeto receptor del mensaje."

^subjectClass!

subjectClass: aSubjectClass
"Setea el nombre de la clase subjetiva a la que pertenece el objeto receptor del mensaje."

subjectClass:= aSubjectClass! !

CLASE Force

'From VisualWorks® Non-Commercial, Release 3.0 of May 8, 1998 on January 9, 2000 at 8:18:07 am'!

Object subclass: #Force
instanceVariableNames: 'name forceOrganizer '
classVariableNames: ''
poolDictionaries: ''
category: 'SubjectiveEnvironment'!

Force comment:
'Force define para cada fuerza los determinantes asociados por clases.

name	<String>	Nombre de la fuerza
forceOrganizer	<ClassOrganizer>	para cada clase, define cuales son los determinantes'!

!Force methodsFor: 'accessing'!

forceOrganizer
"Devuelve el organizador para la fuerza. Este organizador es un ClassOrganizer, definido para las clases y los determinantes de cada clase para la fuerza"

```
^ forceOrganizer.!
```

forceOrganizer: aOrganizer
"Setea el organizador para la fuerza. Este organizador es un ClassOrganizer, definido para las clases y los determinantes de cada clase para la fuerza"

```
forceOrganizer := aOrganizer.!
```

name
"Devuelve el nombre de la fuerza"

```
^ name.!
```

name: aString
"Setea el nombre de la fuerza como aString"

```
name := aString.
^ name.! !
```

!Force methodsFor: 'initialize'!

initialize: aString
"Inicializa las variables de instancia de la fuerza, setea el nombre y crea un organizador vacio, a medida que se definan los determinantes para cada clase, se iran agregando datos al mismo"

```
self name: aString.
self forceOrganizer: (ClassOrganizer new).! !
```

CLASE ForceAdministrator

'From VisualWorks® Non-Commercial, Release 3.0 of May 8, 1998 on January 9, 2000 at 8:18:09 am'!

SimpleListEditor subclass: #ForceAdministrator
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'SubjectiveEnvironment'!

ForceAdministrator comment:
'ForceAdmin is a simple list application, that allows users to add, delete, or modify elements in a list.

Instance Variables:
list <SelectionInList> the list being edited.
currentName <ValueHolder> the current selection in the list, or a new element to be added to the list.
validationBlock <BlockClosure> when a list elemented is added or changed, this block is evaluated to see if the new element is legal.
changedBlock <BlockClosure> evaluated whenever the list is modified.
'!

!ForceAdministrator methodsFor: 'initialize'!

initialize
"Inicializa el administrador de fuerzas"

```
super initialize.
list := SelectionInList new.
list list: SubjectivitySmalltalk getForceNameList.
list selectionIndexHolder onChangeSend: #activateButtons to: self.
list selectionIndexHolder onChangeSend: #updateCurrentName to: self.
currentName := nil asValue.
currentName onChangeSend: #activateButtons to: self.
currentName value: ''.
"El bloque de validacion se arma de la siguiente manera:
        - string ingresada no vacia
      AND - no existe dentro de la coleccion ninguna fuerza con el nombre de la string
      ingresada
"
validationBlock := [:newStr |
                        (newStr isEmpty not) and:
                        [(SubjectivitySmalltalk hasForceNamed: newStr)not.]
                   ].
changedBlock := [nil].! !
```

!ForceAdministrator methodsFor: 'accessing'!

addName
" Toma la string ingresada en el listbox (newForceName), y verifica si se puede o no agregar un objeto Force a la coleccion de fuerzas existentes en el Smalltalk -variable SubjectivitySmalltalk-.
Si la string esta vacia o corresponde al nombre de una fuerza ya existente, se obtiene un mensaje de error.
Sino agrega una fuerza con dicho nombre en SubjectivitySmalltalk. "

```
| newForceName |
newForceName := currentName value.
(validationBlock value: newForceName)
ifTrue:
            [ SubjectivitySmalltalk newForce: newForceName.
              (list list includes: newForceName)
                  ifFalse:
                        [(list list) add: newForceName.
                        changedBlock value].
            ]
ifFalse:
```

```
                    [Dialog warn: 'This is not a legitimate entry for the list. You cannot add a Force
without name or a Force which already exists.' for: builder window.
                    ^self].
        list selection: newForceName.!
```

changedBlock: aBlock

```
        changedBlock := aBlock!
```

changeName

"Cambia en todo el ambiente de Smalltalk el nombre de la fuerza seleccionada. Esto significa la fuerza en s, y todas aquellas opciones e implementaciones relacionadas con la misma"
"Metodologia:
1. Si la fuerza seleccionada no existe como tal, o no se ingreso el nuevo nombre de fuerza, entonces dar un mensaje de error y terminar.
2. En este punto estamos seguros que la fuerza existe como tal. Entonces:
a. Obtengo el objeto fuerza con el nombre dado, de SubjectivitySmalltalk (forceToBeChanged).
b. A dicho objeto le solicito su forceOrganizer, porque alli estan almacenadas todas las clases y sus correspondientes opciones para esta fuerza a eliminar (forceOrganizer).
c. Al forceOrganizer le pido los nombres de todas las clases que implementan esta fuerza. (classList).
d. Para cada una de las clases mencionadas en classList, debo obtener su methodDictionary.
e. Recorrer el methodDictionary buscando solamente los metodos subjetivos. Para cada metodo subjetivo encontrado, debo solicitarle cambie el nombre de la fuerza anterior por el nuevo nombre.
f. Por ultimo cambio el nombre de la fuerza en si misma (forceToBeChanged) de la coleccion de fuerzas existentes en Smalltalk (ForceCollection)
g. Actualizo el listbox que visualiza el usuario.
"

```
        | newForceName oldForceName |
        list selectionIndex = 0 ifTrue: [ Dialog warn: 'Please, select the force you want to change.\'
withCRs.
                                        ^nil.
                                      ].
        oldForceName := (list list) at: list selectionIndex.
        newForceName := currentName value.
        self validationBlock:  [:newStr : oldStr |
                                    (newStr isEmpty not)
                                     and: [SubjectivitySmalltalk hasForceNamed: oldStr.]
                                ].

        (validationBlock value: newForceName value: oldForceName )
        ifTrue:   [ | forceToBeChanged  forceOrganizer classList |
                    (Dialog confirm: 'Do you really want to change the ' , oldForceName, ' force name by
the new name ', newForceName, ' ? \' withCRs initialAnswer: false) ifFalse: [^nil].

                    forceToBeChanged := SubjectivitySmalltalk getForce: oldForceName.
                    forceOrganizer := forceToBeChanged forceOrganizer.
                    classList := forceOrganizer categories.
                    classList do:  [ :className |  |methodDict |
                                        methodDict := (Smalltalk at: className) getMethodDictionary.
                                        methodDict do: [ :method |
                                                            (method class) == CompiledMethod
                                                            ifFalse: [method changeForceName:
(forceToBeChanged name) by: newForceName.
```

```
                                                                ].
                                                        ].
                                                ].
                        forceToBeChanged name: newForceName.
                        Dialog warn: 'Remember that you must review your Force Logic Blocks, and change
the logic associated with the force you have just changed.' for: builder window.
                        (list list includes: oldForceName)
                                ifTrue:
                                        [(list list) at: list selectionIndex put: newForceName.
                                        changedBlock value].
                ]
        ifFalse: [Dialog warn: 'This is not a legitimate entry for the list. You cannot change a Force without
name or a Force which does not exist. Be sure you have selected the force to change and you have
provided the new name for this force.' withCRs for: builder window.
                        ^self].
        list selection: newForceName.
        changedBlock value!
```

currentName

"Devuelve el nombre de la fuerza seleccionada"

```
        ^currentName!
```

deleteName

"Elimina de todo el ambiente de Smalltalk la fuerza seleccionada. Esto significa la fuerza en sÃ, y todos aquellos determinantes e implementaciones relacionadas con la misma"
"Metodologia:
1. Si la fuerza seleccionada / tipeada no existe como tal, entonces dar un mensaje de error y terminar.
2. En este punto estamos seguros que la fuerza existe como tal. Entonces:
        a. Obtengo el objeto fuerza con el nombre dado, de SubjectivitySmalltalk (forceToBeRemoved).
        b. A dicho objeto le solicito su forceOrganizer, porque alli estan almacenadas todas las clases y sus correspondientes opciones para esta fuerza a eliminar (forceOrganizer).
        c. Al forceOrganizer le pido los nombres de todas las clases que implementan esta fuerza. (classList).
        d. Para cada una de las clases mencionadas en classList, debo obtener su methodDictionary.
        e. Recorrer el methodDictionary buscando solamente los metodos subjetivos. Para cada metodo subjetivo encontrado, debo solicitarle que elimine toda implementacion que se encuentre bajo la fuerza a eliminar.
        En caso de estar eliminando la ultima implementacion existente para un determinado selector, debo proceder a eliminar la entrada del methodDictionary para dicho selector, asi como tambien eliminar la entrada en el organization de la clase en cuestion.
        f. Por ultimo elimino la fuerza en si misma (forceToBeRemoved) de la coleccion de fuerzas existentes en Smalltalk (ForceCollection)
        g. Actualizo el listbox que visualiza el usuario.
"

```
        | selectedForceName |
        selectedForceName := currentName value.
        validationBlock :=  [:newStr |
                                (newStr isEmpty not)
                                and: [SubjectivitySmalltalk hasForceNamed: newStr.
                                        ]
                        ].
        (validationBlock value: selectedForceName)
```

```
        ifTrue:   [ | forceToBeRemoved forceOrganizer classList |
                    (Dialog confirm: 'Do you really want to delete the ' , selectedForceName, ' force from
Smalltalk ? \' withCRs initialAnswer: false) ifFalse: [^nil].
                    forceToBeRemoved := SubjectivitySmalltalk getForce: selectedForceName.
                    forceOrganizer := forceToBeRemoved forceOrganizer.
                    classList := forceOrganizer categories.
                    classList do:  [ :className |  |methodDict |
                                methodDict := (Smalltalk at: className) getMethodDictionary.
                                methodDict do: [ :method |
                                            (method class) == CompiledMethod
                                            ifFalse: [method removeForce: selectedForceName.
                                                    method hasImplementations
                                                    ifFalse: [  | selector |
                                                            selector := method who at: 2.
                                                            (method who at: 1) organization
removeElement: selector.
                                                              methodDict removeKey: selector
ifAbsent: [].

                                                                    ].
                                            ].
                                ].
                    ].
                    SubjectivitySmalltalk removeForce: forceToBeRemoved.
                    Dialog warn: 'Remember that you must review your Force Logic Blocks, and delete the
logic associated with the force you have just deleted.' for: builder window.
                    (list list includes: selectedForceName)
                    ifTrue:   [ list list remove: selectedForceName ifAbsent: [].
                              changedBlock value
                            ].
                    ]
        ifFalse: [Dialog warn: 'This is not a legitimate entry for the list. You cannot delete a Force without
name or a Force which does not exist.' for: builder window.
                     ^self
                    ].
        list selectionIndex: 0.
        changedBlock value.!
```

list
"Devuelve la lista de los nombres de las fuerzas definidas en el ambiente"

```
        ^list!
```

list: aList
"Setea la lista de fuerzas"

```
        list list: aList!
```

listHolder: aValueModel

```
        list listHolder: aValueModel!
```

onlyChangeName

```
        | str |
        list selectionIndex = 0 ifTrue:
```

```
            [Dialog warn: 'Please select an entry first' for: builder window.
            ^self].
        str := currentName value.
        (validationBlock value: str) ifFalse:
            [Dialog warn: 'This is not a legitimate entry for the list' for: builder window.
            ^self].
        (list list includes: str) ifFalse:
            [(list list) at: list selectionIndex put: str.
            changedBlock value].
        list selection: str.!
```

validationBlock: aBlock

```
        validationBlock := aBlock! !
```

!ForceAdministrator methodsFor: 'updating'!

activateButtons

```
        "We can't currently do this"!
```

updateCurrentName

```
        ^list selectionIndexHolder value = 0
            ifTrue: [currentName value: '']
            ifFalse: [currentName value: list selection]! !
```

!ForceAdministrator methodsFor: 'drag and drop'!

doDrag: aController
"Drag the currently selected change. Include all available information so that the drop target can use whatever it needs."

```
        | data |
        data := DragDropData new.
        data contextWindow: builder window.
        data contextWidget: aController.
        data contextApplication: self.
        data clientData: list selection.
        (DragDropManager
                withDropSource: DropSource new
                withData: data) doDragDrop!
```

dragEnter: aDragContext
"A drag has entered the editor's list. If the receiver is happy to accept a drag from the target the fill-in the appropriate data and answer that dragging should continue."

```
        self == aDragContext data contextApplication
            ifFalse: [^#dropEffectNone].
        aDragContext dropTarget clientData: self initialDropState.
        ^#dropEffectMove!
```

dragLeave: aDragContext
"A drag has left the editor's list. If the drag contains a change, we must restore the state of the list."

```
        self == aDragContext data contextApplication
```

```
		ifTrue:
			[self restoreListStateFrom: aDragContext dropTarget clientData.
			aDragContext dropTarget clientData: nil].
	^#dropEffectNone!
```

dragOver: aDragContext
"A drag is over the editor's list."

```
	^self == aDragContext data contextApplication
		ifTrue:
			[(self builder componentAt: #list) widget
				showDropFeedbackIn: aDragContext
				allowScrolling: true.
			#dropEffectMove]
		ifFalse:
			[(Object errorSignal
					handle: [:ex| ex returnWith: false]
					do: [validationBlock value: aDragContext data clientData])
				ifTrue: [#dragEffectCopy]
				ifFalse: [#dropEffectNone]]!
```

drop: aDragContext
"A drop has occurred on the list. If its from ourself then reorder the list appropriately."

```
	| view element theList effectiveTargetIndex sourceIndex destIndex |
	(self dragOver: aDragContext) == #dropEffectNone ifTrue:
		[^#dragEffectNone].

	element := aDragContext data clientData.
	theList := list list.

	"Dropping in the receiver itself.  We need to map the list target
	 and drop indices from the displayed list to the underlying list of
	 elements and move the element in the underlying list.
	 First find out where the target really is."
	view := (self builder componentAt: #list) widget.
	effectiveTargetIndex := view elementIndexFor: view controller sensor cursorPoint.

	sourceIndex := theList indexOf: element.
	destIndex := theList indexOf: (theList at: ((effectiveTargetIndex min: theList size) max: 1)).
	sourceIndex = destIndex ifTrue: [^#dragEffectNone].
	sourceIndex ~= 0 ifTrue: [theList removeAtIndex: sourceIndex].
	"If dragging upward then insert before; if dragging down or into then insert after."
	(destIndex >= sourceIndex
	or: [sourceIndex = 0])
		ifTrue:
			[destIndex > theList size
				ifTrue: [theList addLast: element]
				ifFalse: [theList add: element beforeIndex: 1 + destIndex - (sourceIndex
min: 1)]]
		ifFalse:
			[theList add: element beforeIndex: destIndex].
	list selection: element.
	changedBlock value.
	^#dropEffectNone!
```

initialDropState
"Answer the initial state of the list when a drag has begun within its bounds. The state will be used to re-establish the appearance of the list when a drag leaves."

```
| dict view ctrl |
ctrl := (view := (self builder componentAt: #list) widget) controller.
dict := IdentityDictionary new.
dict at: #ctrl put: ctrl.
dict at: #targetIndex put: view targetIndex.
dict at: #effectiveTargetIndex put: (view elementIndexFor: ctrl sensor cursorPoint).
dict at: #hasFocus put: ctrl view hasFocus.
ctrl view hasFocus: true.
^dict!
```

restoreListStateFrom: aDictionary
"Reset the state of the list when a drag has left its bounds."

```
| ctrl |
ctrl := aDictionary at: #ctrl.
ctrl view targetIndex: (aDictionary at: #targetIndex).
ctrl view hasFocus: (aDictionary at: #hasFocus)!
```

wantToDrag: listController
"Answer true if the receiver wants to initiate a drag."

```
    ^list list size > 1! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

ForceAdministrator class
    instanceVariableNames: ''!
```

!ForceAdministrator class methodsFor: 'interface specs'!

changeOnlyWindowSpec
"UIPainter new openOnClass: self andSelector: #changeOnlyWindowSpec"

```
<resource: #canvas>
^#(#FullSpec
    #window:
    #(#WindowSpec
        #label:  'Forces Administrator'
        #bounds: #(#Rectangle 156 292 410 495 ) )
    #component:
    #(#SpecCollection
        #collection: #(
            #(#ActionButtonSpec
                #layout: #(#LayoutFrame 2 0.66667 -35 1 -2 1 0 1 )
                #name: #change
                #model: #onlyChangeName
                #label: 'Change'
                #isDefault: true
                #defaultable: true )
            #(#InputFieldSpec
                #layout: #(#LayoutFrame 2 0 -70 1 -2 1 -40 1 )
```

```
                    #model: #currentName )
                #(#SequenceViewSpec
                    #layout: #(#LayoutFrame 2 0 2 0 -2 1 -75 1 )
                    #name: #list
                    #model: #list
                    #useModifierKeys: true
                    #selectionType: #highlight ) ) ) )!
```

windowSpec

```
"UIPainter new openOnClass: self andSelector: #windowSpec"

    <resource: #canvas>
    ^#(#FullSpec
        #window:
        #(#WindowSpec
            #label: 'Forces Administrator'
            #bounds: #(#Rectangle 421 319 675 522 ) )
        #component:
        #(#SpecCollection
            #collection: #(
                #(#ActionButtonSpec
                    #layout: #(#LayoutFrame 2 0 -35 1 -2 0.333333 0 1 )
                    #name: #add
                    #model: #addName
                    #label: 'Add'
                    #defaultable: true )
                #(#ActionButtonSpec
                    #layout: #(#LayoutFrame 2 0.333333 -35 1 -2 0.666667 0 1 )
                    #name: #delete
                    #model: #deleteName
                    #label: 'Delete'
                    #defaultable: true )
                #(#ActionButtonSpec
                    #layout: #(#LayoutFrame 2 0.666667 -35 1 -2 1 0 1 )
                    #name: #change
                    #model: #changeName
                    #label: 'Change'
                    #defaultable: true )
                #(#InputFieldSpec
                    #layout: #(#LayoutFrame 2 0 -70 1 -2 1 -40 1 )
                    #model: #currentName )
                #(#SequenceViewSpec
                    #properties:
                    #(#PropertyListDictionary
                        #dragOkSelector #wantToDrag:
                        #dragEnterSelector #dragEnter:
                        #dragOverSelector #dragOver:
                        #dragStartSelector #doDrag:
                        #dropSelector #drop:
                        #dragExitSelector #dragLeave: )
                    #layout: #(#LayoutFrame 2 0 2 0 -2 1 -75 1 )
                    #name: #list
                    #model: #list
                    #useModifierKeys: true
                    #selectionType: #highlight ) ) ) )! !
```

CLASE ForceLogicDefinition

'From VisualWorks® Non-Commercial, Release 3.0 of May 8, 1998 on January 9, 2000 at 8:18:12 am'!

Object subclass: #ForceLogicDefinition
instanceVariableNames: 'logicBlocks '
classVariableNames: ''
poolDictionaries: ''
category: 'SubjectiveEnvironment'!

ForceLogicDefinition comment:
'ForceLogicDefinition contiene los bloques de logica de fuerza asociados a cada clase o metodo.

logicBlocks <Dictionary> Diccionario de bloques de logica de fuerzas'!

!ForceLogicDefinition methodsFor: 'initialize'!

initialize
“Inicializa el diccionario de bloques de logicas de fuerzas"

```
logicBlocks := Dictionary new.! !
```

!ForceLogicDefinition methodsFor: 'removing'!

deleteAllBlocksInClass: aClass
"Elimina todos los bloques de lÃ³gica de fuerza definidos para la clase aClass."

```
(logicBlocks includesKey: aClass)
        ifTrue: [ logicBlocks removeKey: aClass ifAbsent: [].
      ]
        ifFalse: [ ].!
```

deleteBlockIn: aClass and: aSelector
"Elimina el bloque de logica de fuerza definido para el metodo aSelector de la clase aClass.
Si aSelector es nil, entonces elimina el bloque de logica de fuerza definido a nivel de clase."

```
(logicBlocks includesKey: aClass)
        ifTrue: [ (aSelector == nil )
                ifTrue: [ (logicBlocks at: aClass) removeKey: aClass ifAbsent: [].
                        ]
                ifFalse: [ (logicBlocks at: aClass) removeKey: aSelector ifAbsent: [].
                        ]
              ]
        ifFalse: [ ].! !
```

!ForceLogicDefinition methodsFor: 'adding'!

addBlock: aNewBlock in: aClass and: aSelector
"Agrega un bloque de logica de fuerza para una clase dada. Si aSelector es nil, significa que es la Logica de Fuerzas para la Clase"

```
(logicBlocks includesKey: aClass)
        ifFalse: [logicBlocks at: aClass put: (Dictionary new)].

(aSelector == nil)
        ifTrue:   [(logicBlocks at: aClass) at: aClass put: aNewBlock]
        ifFalse: [(logicBlocks at: aClass) at: aSelector put: aNewBlock].! !
```

!ForceLogicDefinition methodsFor: 'accessing'!

getBlockIn: aClass and: aSelector
"Devuelve el bloque de logica de fuerza de una clase dada. Es utilizado en el SubjBrowser para mostrar el codigo de la logica de fuerzas en el Pane correspondiente. Si no tiene logica definida, retorna un template del bloque requerido. Si aSelector es nil, devuelve la Logica definida a nivel de Clase, sino a nivel de metodo"

```
(logicBlocks includesKey: aClass)
        ifTrue:[ ((logicBlocks at: aClass) includesKey: aSelector)
                                ifTrue:   [^(logicBlocks at: aClass)  at: aSelector]
                                ifFalse: [^ForceLogicDefinition template]
                ]
        ifFalse: [^ForceLogicDefinition template].!
```

getInfluenceForceFor: anExecutionContext
"Retorna la Fuerza y el Determinante predomenantes para un objeto en un momento determinado.
El parametro anExecutionContext contiene toda la informacion necesaria para realizar esta evaluacion."

```
| lb compiledLogicBlock dict |

"Si no hay una clave con el nombre de la Clase ==> no hay definida Logica de Fuerzas"
(logicBlocks includesKey: (anExecutionContext subjectClass name))
        ifFalse:[^nil].

"Si no hay una clave con el nombre del selector del metodo ==> no hay Logica de Fuerzas a nivel metodo"
dict := logicBlocks at: (anExecutionContext subjectClass name).
(dict includesKey: (anExecutionContext selector))
        "Devuelvo la Logica de Fuerzas a nivel Clase (por las dudas controlo si esta definida)"
        ifFalse: [(dict includesKey: (anExecutionContext subjectClass name))
                        ifFalse: [lb := nil]
                        ifTrue:   [lb := dict at: (anExecutionContext subjectClass name)]
                ]
        "Devuelvo la Logica de Fuerzas a nivel metodo"
        ifTrue: [lb := dict at:(anExecutionContext selector)].

 (lb == nil)
        ifTrue: [ ^nil ].

compiledLogicBlock := Compiler evaluate: lb for: (anExecutionContext subject) logged: false.
compiledLogicBlock method outerMethod sourcePointer: (lb asString).
compiledLogicBlock method outerMethod mclass: (anExecutionContext subjectClass).
```

```
    ^( compiledLogicBlock value: anExecutionContext ).!
```

getSelectorFor: aText in: aClass
"Obtiene el selector del metodo cuyo texto de logica de fuerzas es aText, revisando en el diccionario de bloques de logica de fuerzas de la clase aClass, la clave del mismo es el selector del metodo"

```
        (logicBlocks includesKey: aClass)
                ifFalse: [^nil]
                ifTrue:  [ | dict |
                           dict := logicBlocks at: aClass.
                           dict keysAndValuesDo: [:aKey :aValue | aText = aValue ifTrue: [^aKey]].
                           ^nil.
                         ].! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

ForceLogicDefinition class
        instanceVariableNames: ''!
```

!ForceLogicDefinition class methodsFor: 'browsing'!

template
"Muestra el template para cualquier definicion de logica de fuerzas"

```
        ^ '[:anExecutionContext |
                | anInfluenceForce |
                anInfluenceForce := InfluenceForce new.
                " >> Put your logic definition here <<
                You can define this logic using the values specified in anExecutionContext.
                For more information, see the ExecutionContext class comment.
                "
                anInfluenceForce force: "aForce".
                anInfluenceForce determinant: "aDeterminant"
                anInfluenceForce].'.! !
```

CLASE InfluenceForce

'From VisualWorks® Non-Commercial, Release 3.0 of May 8, 1998 on January 9, 2000 at 8:18:14 am'!

Object subclass: #InfluenceForce

```
        instanceVariableNames: 'force determinant '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'SubjectiveEnvironment'!
```

InfluenceForce comment:
'InfluenceForce define la fuerza y el deteminante influyente en un momento determinado, durante la recepcion de un mensaje

force	<String>	Nombre de la fuerza (por default, uno de los siguientes valores: ["State"|"Sender"| "Content"])
determinant	<String>	Nombre del Determinante, segun la fuerza.'!

!InfluenceForce methodsFor: 'accessing'!

determinant
"Devuelve el determinante influyente sobre un objeto"

```
^determinant.!
```

determinant: aDeterminant
"Setea el determinante influyente sobre un objeto"

```
determinant := aDeterminant.!
```

force
"Devuelve la fuerza influyente sobre un objeto"

```
^force.!
```

force: aForce
"Setea la fuerza influyente sobre un objeto"

```
force := aForce.! !
```

CLASE NotifierView

'From VisualWorks® Non-Commercial, Release 3.0 of May 8, 1998 on January 9, 2000 at 8:18:16 am'!

View subclass: #NotifierView
```
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'SubjectiveEnvironment'!
```

NotifierView comment:
'NotifierView is the class for handling a user-visible interruption. Its primary purpose is to create notifier windows.
'!

"-- "!

NotifierView class
```
instanceVariableNames: ''!
```

!NotifierView class methodsFor: 'class initialization'!

initialize
"Make the receiver the default Notifier."

```
Notifier beDevelopment!
```

obsolete
"Deinstall the receiver as the default Notifier."

```
self == NotifierView ifTrue: [Notifier beDeployed].
super obsolete! !
```

!NotifierView class methodsFor: 'instance creation'!

openContext: haltContext label: aString proceedable: aBoolean
"Create and schedule an instance of me viewing a Debugger on haltContext. The view will be labeled with aString, and will show a short sender stack."

```
| displayPoint contentsString |

"undo all grabs for event driven"
InputState default ungrabBecauseOfError.

"Make sure that controllers without views are removed"
ScheduledControllers removeInvalidControllers.
contentsString := self shortStackFor: haltContext ofSize: 5.
displayPoint :=
        (ScheduledControllers activeControllerProcess ~~ Processor activeProcess
                or: [ScheduledControllers activeController == nil])
                ifTrue: [Screen default bounds center]
                ifFalse: [| view |
                                view := ScheduledControllers activeController view.
                                view displayBox center].
self openDebugger: (self debuggerClass context: haltContext proceedable: aBoolean)
        contents: contentsString
        label: aString
        displayAt: displayPoint.
thisContext unwindUpTo: haltContext.
Processor activeProcess suspend!
```

openException: exception
"Create and schedule an instance of me viewing a Debugger on haltContext. The view will be labeled with aString, and shows a short sender stack."

```
| displayPoint isInterrupt |

"undo all grabs for event driven"
InputState default ungrabBecauseOfError.

"Make sure that controllers without views are removed"
ScheduledControllers removeInvalidControllers.
displayPoint :=
        (ScheduledControllers activeControllerProcess ~~ Processor activeProcess
                or: [ScheduledControllers activeController == nil])
                ifTrue: [Screen default bounds center]
                ifFalse: [| view |
                                view := ScheduledControllers activeController view.
                                view displayBox center].
isInterrupt := exception initialContext method homeMethod ==
                                (ControlManager compiledMethodAt: #interruptName:).
self openDebugger: (self debuggerClass new
                                process: Processor activeProcess
```

```
                                        context: exception parameter
                                        interrupted: isInterrupt
                                        proceedable: exception willProceed)
                contents: (self shortStackFor: exception parameter ofSize: 5)
                label: exception errorString
                displayAt: displayPoint.
        thisContext unwindUpTo: exception parameter.
        Processor activeProcess suspend! !
```

!NotifierView class methodsFor: 'private'!

debuggerClass

```
        ^SubjectivityDebugger!
```

logErrorFor: aDebugger label: label

```
"Log the error to a file in case the notifier can't come up."
"If you want to enable logging of notifiers, change the first 'false' to 'true', which will cause logging to be enabled.  Then edit the  file name (the default is 'visual.err') to whatever file you want to write the logs to.  Make sure that the file is writable, and that there is enough disk space.  If there is any error writing the log file, the error will be silently ignored--otherwise the system would go into infinite recursion and run out of memory."

        false ifTrue:
                [Object errorSignal
                        handle: [:ex | ex return]
                        do: [| file |
                                file := 'visual.err' asFilename appendStream.
                                [file cr; cr; nextPutAll: label; cr.
                                aDebugger contextList do:
                                        [:c | c printOn: file. file cr]]
                                valueNowOrOnUnwindDo: [file close]]].!
```

openDebugger: aDebugger contents: aString1 label: aString2 displayAt: aPoint

```
        | box text y builder label actions specs width height copyStack correctIt baseFraction |

        self logErrorFor: aDebugger label: aString2.

        aDebugger prepareForErrorCondition.
        builder := UIBuilder new.
        builder windowOn: aDebugger label: 'Exception'.
        aDebugger label: aString2.
        text := aString2 asText.
        text emphasizeFrom: 1 to: text size with:
                        (Screen default colorDepth < 4
                                ifTrue: [#bold]
                                ifFalse: [Array with: #bold with: #color->(ColorValue red: 0.8 green: 0
blue: 0)]).
        text := ComposedText withText: text style: nil compositionWidth: 250.
        y := text bounds height // 2 max: 16.
        label := LabelSpec new hasCharacterOrientedLabel: false.
        label setLabel: builder policy alertIcon.
        label layout: (AlignmentOrigin new
                                leftOffset: 48; topOffset: y+16;
                                leftAlignmentFraction: 1;
```

```
                        topAlignmentFraction: 0.5).
builder add: label.
label := LabelSpec new hasCharacterOrientedLabel: false.
label setLabel: text.
label layout: (AlignmentOrigin new
                        leftOffset: 56; topOffset: y+16;
                        leftAlignmentFraction: 0;
                        topAlignmentFraction: 0.5).
y := y * 2 + 32.
builder add: label.
actions := OrderedCollection
        with: 'Debug' ->
                [Debugger
                        openFullViewOn: aDebugger
                        label: aString2.
                builder window controller close]
        with: 'Proceed' -> [aDebugger proceed]
        with: 'Terminate' -> [builder window controller close].
specs := OrderedCollection new.
1 to: actions size do:
        [:i | | action |
        action := actions at: i.
        specs add: (ActionButtonSpec
                        model: action value
                        label: action key
                        layout: (LayoutFrame new
                                leftFraction: i-1/actions size;
                                rightFraction: i / actions size;
                                topOffset: y;
                                bottomOffset: y)).
        specs last defaultable: true.
        action key = 'Debug'
                ifTrue:
                        [specs last isDefault: true.
                        builder add: specs last.
                        builder keyboardProcessor setActive: builder wrapper widget controller]
                ifFalse: [builder add: specs last].
        (action key = 'Proceed' and: [aDebugger mayProceed not])
                ifTrue: [builder wrapper disable]].
y := y + builder wrapper preferredBounds height.
specs do:
        [:spec |
        spec layout bottomOffset: y].
baseFraction := actions size - 1 / 2 / actions size.
copyStack := (ActionButtonSpec
                        model: [aDebugger copyStack]
                        label: 'Copy stack'
                        layout: (LayoutFrame new
                                leftFraction: baseFraction;
                                rightFraction: baseFraction + (1/actions size);
                                topOffset: y;
                                bottomOffset: y + builder wrapper preferredBounds height)).
copyStack defaultable: true.
builder add: copyStack.
((aDebugger interruptedContext isKindOf: Context)
and: [aDebugger interruptedContext selector == #doesNotUnderstand:])
```

```
                ifTrue:
                        [correctIt := (ActionButtonSpec
                                        model: [aDebugger correctSpelling]
                                        label: 'Correct it...'
                                        layout: (LayoutFrame new
                                                leftFraction: baseFraction + (1/actions size);
                                                rightFraction: baseFraction + (2/actions size);
                                                topOffset: y;
                                                bottomOffset: y + builder wrapper preferredBounds
height)).
                        correctIt defaultable: true.
                        builder add: correctIt].
        y := y + builder wrapper preferredBounds height.
        label := LabelSpec new hasCharacterOrientedLabel: false.
        label setLabel: aString1 asComposedText.
        label layout: (AlignmentOrigin new
                                leftFraction: 0.5 offset: 0;
                                topFraction: 0.5 offset: y // 2;
                                leftAlignmentFraction: 0.5;
                                topAlignmentFraction: 0.5).
        builder add: label.
        width := 320 max: label getLabel bounds width+12.
        height := y + label getLabel bounds height + 6.
        box := 0@0 corner: width@height.
        box := box align: box center with: aPoint.
        builder openIn: box.
        ScheduledControllers launchBaseProcess!
```

shortStackFor: aContext ofSize: anInteger
"Answer a string with the simple descriptions of anInteger number of non-nil stackframes starting with aContext"

```
        | shortStackStream ctx |
        shortStackStream := WriteStream on: (String new: 400).
        ctx := aContext.
        anInteger timesRepeat:
                [ctx isNil ifFalse:
                        [shortStackStream nextPutAll: (ctx printString contractTo: 50); cr.
                        ctx := ctx sender]].
        shortStackStream position > 0 ifTrue: [shortStackStream skip: -1].
        ^shortStackStream contents! !

NotifierView initialize!
```

CLASE NullCompiledMethod

'From VisualWorks® Non-Commercial, Release 3.0 of May 8, 1998 on January 9, 2000 at 8:18:20 am'!

```
SubjectiveMethod variableSubclass: #NullCompiledMethod
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'SubjectiveEnvironment'!

NullCompiledMethod comment:
'Representa al elemento Neutro dentro de la jerarquÃa de CompiledMethods. Permite definir
un metodo que simplemente hace un ^super de sÃ mismo. No tiene variables de instancias.'!


!NullCompiledMethod methodsFor: 'private'!

initializeInClass: aClass withFullSelector: aFullSelector withSelector: aSelector forMethod:
aCompiledMethod classified: aProtocol
" Inicializa los bytecodes y lo coloca en el diccionario de metodos de manera correcta. "

	mclass := aClass.
	self bytecodesFor: aCompiledMethod.
	aClass removeSelector: aSelector.
	^self.! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

NullCompiledMethod class
	instanceVariableNames: ''!

!NullCompiledMethod class methodsFor: 'templates'!

executionTemplate: aSelector
"Retorna el texto del codigo que corresponde a un CompiledMethod nulo, que invoca al metodo
'aSelector' de la superclase."

	^Text fromString: (aSelector,'
	^super ' , aSelector).! !

!NullCompiledMethod class methodsFor: 'creating'!

newForClass: aClass withSelector: aFullSelector classified: aProtocol notifying: aController
forSubjectiveMethod: aSubjectiveMethod withSelector: aSelector
"Crea un NullCompiledMethod, dejando el MethodDictionary el subjectiveMethod si es que existia"

	| aNullCompiledMethod|

	aNullCompiledMethod:= super newForClass: aClass withSelector: aFullSelector classified:
aProtocol notifying: aController.

	(aSelector isNil)
		ifFalse:[ aClass addSelector: aSelector withMethod: aSubjectiveMethod].
	^aNullCompiledMethod! !
```

CLASE SubjectiveMethod

'From VisualWorks® Non-Commercial, Release 3.0 of May 8, 1998 on January 9, 2000 at 8:18:24 am'!

CompiledMethod variableSubclass: #SubjectiveMethod
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'SubjectiveEnvironment'!

SubjectiveMethod comment:
'SubjectiveMethod permite definir el comportamiento comÃºn entre las clases SubjectivityCompiledMethod y NullCompiledMethod, relaciona principalmente con la creacion de objetos de esta clase y la compilacion de los template method asociados a cada clase.

No tiene variables de instancia.

'!

!SubjectiveMethod methodsFor: 'private'!

bytecodesFor: aCompiledMethod
"Inicializa los bytecodes propios del SubjectiveMethod, segun se correspondan al selector aSelector para el compiled Method aCompiledMethod"

```
"Le copio las variables de instancia"
1 to: (aCompiledMethod class instVarNames size) do:
        [ :i| self instVarAt: i put: (aCompiledMethod instVarAt:i)
        ].
"Le copio el codigo"
1 to: (aCompiledMethod basicSize) do:
        [:i| self basicAt:i put: (aCompiledMethod basicAt: i)
        ].
self sourcePointer: (aCompiledMethod sourcePointer).!
```

initializeInClass: aClass withFullSelector: aFullSelector withSelector: aSelector forMethod: aCompiledMethod classified: aProtocol
"inicializa los bytecodes correspondientes al compiledMethod interno y lo coloca en el methodDictionary en forma correcta"

```
^self subclassResponsibility.! !
```

!SubjectiveMethod methodsFor: 'accesing'!

changeInMethodDictionary: aSelector classified: aProtocol
"Elimina del method dictionary el compiledMethod asociado al selector, colocando en su lugar al SubjectiveMethod"

```
mclass removeSelector: aSelector.
mclass addSelector: aSelector withMethod: self.
mclass organization classify: aSelector under: aProtocol.
^self.! !
```

"-- "!

SubjectiveMethod class
	instanceVariableNames: "!

!SubjectiveMethod class methodsFor: 'creating'!

newForClass: aClass withSelector: aFullSelector classified: aProtocol notifying: aController

	| temporaryCode temporarySelector temporaryCompiledMethod|
	"Compilo el codigo que corresponda al template y crea un metodo con ese compiledMethod"

	temporaryCode := self executionTemplate: aFullSelector.
	temporarySelector := aClass compile: temporaryCode
	                                                classified: aProtocol
	                                                notifying: aController.
	"Obtengo el CompiledMethod resultante de la compilacion"
	temporaryCompiledMethod := aClass compiledMethodAt: temporarySelector.

	"Armo el methodo partir del CompiledMethod"
	^(super new: (temporaryCompiledMethod basicSize))

initializeInClass: aClass withFullSelector: aFullSelector withSelector: temporarySelector forMethod: temporaryCompiledMethod classified: aProtocol.! !

!SubjectiveMethod class methodsFor: 'templates'!

executionTemplate: aSelector
"Retorna el texto del codigo que le corresponde a los bytecodes."

	^self subclassResponsibility.! !

CLASE SubjectivitvBehavior

'From VisualWorks® Non-Commercial, Release 3.0 of May 8, 1998 on January 9, 2000 at 8:18:26 am'!

Object subclass: #SubjectivityBehavior
	instanceVariableNames: 'forceCollection forceLogicDefinition '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'SubjectiveEnvironment'!

SubjectivityBehavior comment:
'Subjectivity Behavior contiene la informacion necesaria para manejar el ambiente subjetivo: las fuerzas existentes en el ambiente y definiciones de logicas de fuerzas definidas para cada clase o metodo.

forceCollection	<OrderedCollection>	Contiene las fuerzas definidas en el ambiente
forceLogicDefinition	<ForceLogicDefinition> Contiene la definicion de la logica de fuerza asociada a cada clase o metodo.
'!

!SubjectivityBehavior methodsFor: 'initialize'!

initialize
"Inicializa la coleccion de fuerzas y la definicion de logicas de fuerzas"

```
forceCollection := OrderedCollection new.
forceLogicDefinition  := ForceLogicDefinition new initialize.! !
```

!SubjectivityBehavior methodsFor: 'accessing-Force'!

getDeterminantListForForce: aForce intoClass: aClass
" Devuelve una lista con todos los determinantes definidos para la fuerza aForce en la clase aClass"

```
| forceOrganization |

forceOrganization := self getForceOrganizationFor: aForce.
^(forceOrganization listAtCategoryNamed: (aClass name)) asList.!
```

getForce: aString
" Devuelve la primer Fuerza cuyo nombre es aString "

```
^forceCollection detect: [ :force | force name = aString]  ifNone: [] .!
```

getForceNameList
" Devuelve una lista con los nombres de todas las Fuerzas "

```
^(forceCollection collect: [:force | force name ] ) asList.!
```

getForceOrganizationFor: aString
" Devuelve el organizador para la Fuerza cuyo nombre es aString "

```
| force org |

force := forceCollection detect: [: f | f name = aString] ifNone: [nil].
(force == nil)
        ifTrue:   [org := ClassOrganizer new]
        ifFalse: [org := force forceOrganizer].

^org.!
```

hasForceNamed: aString
" Devuelve si contiene una Fuerza con el nombre aString "

```
|newCollection|

newCollection:= forceCollection select: [ :force | force name == aString ].
^(newCollection isEmpty not).!
```

newForce: aString
" Agrega la fuerza aForce a la coleccion de Fuerzas "

```
| force |

force := Force new initialize: aString.
forceCollection add: force.!
```

removeForce: aForce
" Borra la Fuerza aForce de la coleccion "

```
forceCollection remove: aForce ifAbsent: [ ].! !
```

!SubjectivityBehavior methodsFor: 'accessing-ForceLogic'!

addForceLogicDefinition: aText in: aClassName
" Guarda el texto de la definicion de la Logica de Fuerzas dentro de la Clase aClassName"

```
forceLogicDefinition addBlock: aText in: aClassName and: aClassName.!
```

addForceLogicDefinition: aText in: aClassName and: aSelector
" Guarda el texto de la definicion de la Logica de Fuerzas dentro de la Clase aClassName y el selector aSelector"

```
forceLogicDefinition addBlock: aText in: aClassName and: aSelector.!
```

deleteAllBlocksInClass: aClass
"Elimina todos los bloques de logica de fuerza definidos para la clase aClass."

```
forceLogicDefinition deleteAllBlocksInClass: aClass.!
```

deleteBlockIn: aClass and: aSelector
"Elimina el bloque de logica de fuerza definido para el metodo aSelector de la clase aClass.
Si aSelector es nil, elimina el bloque de logica de fuerza definido a nivel de clase."

```
forceLogicDefinition deleteBlockIn: aClass and: aSelector!
```

getForceLogicDefinitionFor: aClassName
" Devuelve la definicion de la Logica de Fuerza para la Clase aClassName"

```
^(forceLogicDefinition getBlockIn: aClassName and: aClassName).!
```

getForceLogicDefinitionFor: aClassName and: aSelector
" Devuelve la definicion de la Logica de Fuerza para la Clase aClassName y selector aSelector"

```
^(forceLogicDefinition getBlockIn: aClassName and: aSelector).!
```

getInfluenceForceFor: anExecutionContext
" Evalua el bloque de logica de Fuerzas para el objeto, teniendo en cuenta la informacion de anExecutionContext y devuelve un objeto de la clase InfluenceForce con la fuerza y el determinante predominantes "

```
^forceLogicDefinition getInfluenceForceFor: anExecutionContext.!
```

getSelectorFor: aText in: aClass
"Devuelve el selector correspondiente al bloque de logica de fuerzas aText"

```
^forceLogicDefinition getSelectorFor: aText in: aClass.! !
```

CLASE SubiectivitvBrowser

'From VisualWorks(r) Non-Commercial, Release 3.0 of May 8, 1998 on January 9, 2000 at 8:18:30 am'!

Browser subclass: #SubjectivityBrowser
instanceVariableNames: 'forceButton forceButtonHolder forceSelector forceDeterminantSelector forceOrganization forceLogic forceList forceDeterminantList forceLogicList forceLogicSelector '
classVariableNames: ''
poolDictionaries: ''
category: 'SubjectiveEnvironment'!

SubjectivityBrowser comment:
'SubjectivityBrowser es un Browser que permite mostrar ademÃ¡s los metodos subjetivos, fuerzas y determinantes, con el fin que el programador pueda trabajar con subjetividad de manera amigable.

Instance Variables:

forceButton	<ClassOfVariable>	description of variable''s function
forceButtonHolder	<ClassOfVariable>	description of variable''s function
forceSelector	<ClassOfVariable>	description of variable''s function
forceDeterminantSelector	<ClassOfVariable>	description of variable''s function
forceOrganization	<ClassOfVariable>	description of variable''s function
forceLogic	<ClassOfVariable>	description of variable''s function
forceList	<ClassOfVariable>	description of variable''s function
forceDeterminantList	<ClassOfVariable>	description of variable''s function
forceLogicList	<ClassOfVariable>	description of variable''s function
forceLogicSelector	<ClassOfVariable>	description of variable''s function'!

!SubjectivityBrowser methodsFor: 'determinant list'!

forceDeterminant

"Devuelve el determinante que se encuentra seleccionado"

```
^forceDeterminantSelector!
```

forceDeterminant: selection
"Setea el determinante que se encuentra seleccionado con el valor selection. En base a este debe refrescar el texto del metodo"

```
forceDeterminantSelector := selection.
self newText.!
```

forceDeterminantMenu
"Answer a Menu of operations on message Force Determinant to be displayed when the operate menu button is pressed."

```
((forceSelector ~= nil) and: [(className ~= nil)])
        ifTrue:  [^ Menu labels: 'add...\delete...' withCRs
                        values: #(addForceDeterminant deleteForceDeterminant)
                ]
        ifFalse: [^nil].!
```

newForceDeterminantList: aClassName
"Obtiene la lista de determinantes para la clase aClassName y la fuerza que se encuentra seleccionada y la despliega en el pane de determinantes"

```
	| list |

	(aClassName == nil)
		ifTrue:   [list := List new]
		ifFalse: [list := (forceOrganization listAtCategoryNamed: aClassName) asList.].

	"Remove interested from selection holder because sending #list:
	causes an unnecessary update. Restore interested afterward."

	[self forceDeterminantList selectionIndexHolder retractInterestsFor: self.
	 forceDeterminantList list: list]
		valueNowOrOnUnwindDo:
			[self forceDeterminantList selectionIndexHolder onChangeSend:
#changedForceDeterminant to: self].

	forceDeterminantList selection: forceDeterminantSelector.! !
```

!SubjectivityBrowser methodsFor: 'force list'!

force

```
	^forceSelector.!
```

force: selection
"Set the currently force selector and actualice the force option list"

```
	| currentClassName |

	forceSelector := selection.

	forceOrganization := SubjectivitySmalltalk getForceOrganizationFor: forceSelector.

	(selection == nil)
		ifTrue:   [currentClassName := nil]
		ifFalse: [currentClassName := className].

	self newForceDeterminantList: currentClassName.
	self forceDeterminant: nil.!
```

forceMenu
"Answer a Menu of operations on message Force to be displayed when the operate menu button is pressed."

```
	(forceButton)
		 ifFalse:  [^ Menu labels: 'ABM de fuerzas...' withCRs
				values: #(openForceAdministrator)
			]
		ifTrue: [^nil].!
```

newForceList: aBoolean
"Set the currently force list."

```
	| list |

	(aBoolean)
		ifTrue:   [list := List new]
		ifFalse: [list := SubjectivitySmalltalk getForceNameList].

	["Remove interested from selection holder because sending #list:
	causes an unnecessary update. Restore interested afterward."
	self forceList selectionIndexHolder retractInterestsFor: self.
	forceList list: list]
		valueNowOrOnUnwindDo: [self forceList selectionIndexHolder onChangeSend:
#changedForce to: self].! !
```

!SubjectivityBrowser methodsFor: 'changing'!

changedForce
"Actualiza la lista de fuerzas"

```
	self force: (forceList selection).!
```

changedForceDeterminant
"Actualiza la lista de determinantes"

```
	self forceDeterminant: forceDeterminantList selection.!
```

changedForceLogic
"Actualiza el check para la logica de fuerzas"

```
	self forceLogicSelector: (forceLogicList selection).! !
```

!SubjectivityBrowser methodsFor: 'aspects'!

forceButtonHolder
"Devuelve el contenedor del radio boton de fuerzas (normal o subjetivo)"

```
	^forceButtonHolder!
```

forceDeterminantList
"Devuelve la lista de determinantes"

```
	^forceDeterminantList!
```

forceDeterminantMenuHolder
"Devuelve el menu para el pane de determinantes"

```
	^[self forceDeterminantMenu]!
```

forceList
```
	"Devuelve la lista de fuerzas"

	^forceList!
```

forceLogicList
"Devuelve el check para la logica de fuerzas"

```
^forceLogicList!
```

forceLogicMenuHolder
"Devuelve el menu para el pane de Logica de Fuerzas"

```
^[self forceLogicMenu]!
```

forceMenuHolder
"Devuelve el menu para el pane de fuerzas"

```
^[self forceMenu]! !
```

!SubjectivityBrowser methodsFor: 'initialize-release'!

initialize
"Inicializa las variablas de instancia del SubjBrowser"

```
super initialize.

forceButton := true.
forceButtonHolder := AspectAdaptor subject: self sendsUpdates: true.
forceButtonHolder accessWith: #forceButton assignWith: #forceButton:.

forceLogic := nil.
forceLogicList := SelectionInList with: (List new: 1).
forceLogicList selectionIndexHolder onChangeSend: #changedForceLogic to: self.

forceList := SelectionInList with: (List new).
forceList selectionIndexHolder onChangeSend: #changedForce to: self.

forceDeterminantList := SelectionInList with: (List new).
forceDeterminantList selectionIndexHolder onChangeSend: #changedForceDeterminant to: self.!
```

on: anOrganizer
"Set the receiver to be a browser on the system class organization anOrganizer."

```
| currentForceList |

super on: anOrganizer.
currentForceList := SubjectivitySmalltalk getForceNameList.
forceList := SelectionInList with: currentForceList.
currentForceList size >= 1 ifTrue:
        [forceList selection: currentForceList first].
forceButton := true.! !
```

!SubjectivityBrowser methodsFor: 'norm/subj switch'!

forceButton
"Devuelve la opcion del boton de fuerzas (normal o subjetivo)"

```
^forceButton!
```

forceButton: aBoolean
"Setea la opcion del boton de fuerzas (normal o subjetivo). Luego actualiza los panes de metodos, determinantes, texto y logica de fuerzas"

```
	self changeRequest ifFalse: [ ^self changed: #forceButton].
	forceButton := aBoolean.
	self changed: #forceButton.

	self newForceLogicList.
	self forceLogicSelector: nil.! !
```

!SubjectivityBrowser methodsFor: 'forceLogic'!

forceLogic
"Devuelve el texto del bloque de logica de fuerzas"

```
	^forceLogic!
```

forceLogic: aForceLogicDefinition
"Setea el texto del bloque de logica de fuerzas y se lo pasa a SubjectivitySmalltalk"

```
	forceLogic := aForceLogicDefinition.
	SubjectivitySmalltalk addForceLogicDefinition: aForceLogicDefinition in: (self selectedClass
name) and: selector.
	self newForceList: true.
	self newForceOptionList: nil.
	self newText.!
```

forceLogicMenu
"El pane de logica de fuerzas no tiene menu"

```
	^nil.!
```

forceLogicSelector
"Devuelve el selector de la logica de fuerzas, si esta chequeada, devuelve Logica de Fuerzas, sino nil"

```
	^forceLogicSelector!
```

forceLogicSelector: aSelector
"Setea el selector de la logica de fuerzas"

```
	forceLogicSelector := aSelector.
	self newForceList: ((aSelector ~= nil) or: [forceButton]).
	self force: nil.!
```

newForceLogicList
"Crea la lista para el pane de logica de fuerzas, con un unico elemento que se llama Logica de Fuerzas"

```
	| currentForceLogicList |

	 currentForceLogicList := List new:1.
	((forceButton == false) and: [className ~=  nil])
		ifTrue: [ currentForceLogicList add: 'Logica de Fuerzas'.].

	[self forceLogicList selectionIndexHolder retractInterestsFor: self.
	self forceLogicList list: currentForceLogicList]
```

```
	valueNowOrOnUnwindDo:
		[ self forceLogicList selectionIndexHolder onChangeSend: #changedForceLogic to: self ].!

newForceLogicList: aBoolean
"Setea la lista de logica de fuerzas de acuerdo al valor de aBoolean"

	| currentForceLogicList|

	(aBoolean)
		ifTrue:   [currentForceLogicList := List new]
		ifFalse: [currentForceLogicList := List new add: 'Force Logic'].

	[self forceLogicList selectionIndexHolder retractInterestsFor: self.
	self forceLogicList list: currentForceLogicList]
	valueNowOrOnUnwindDo:
		[ self forceLogicList selectionIndexHolder onChangeSend: #changedForceLogic to: self ].!
!

!SubjectivityBrowser methodsFor: 'private-force'!

openForceAdministrator
"Abre el administrador de fuerzas"

	ForceAdministrator open.
	self newForceList: false.! !

!SubjectivityBrowser methodsFor: 'private-determinant'!

addForceDeterminant
"Da de alta un determinante en el browser como en la Fuerza y en todos los
SubjectivityCompiledMethods de la clase agrega el metodo nulo bajo dicho determinante"

	| determinantName |

	((forceSelector == nil) or: [className == nil])
		ifTrue: [^nil].
	determinantName :=
		Dialog
			request: 'Determinant Name'
			initialAnswer: ParagraphEditor currentSelection.
	"El nombre del determinate no puede ser nulo"
	(determinantName = '' )
		ifTrue: [^nil].

	((forceOrganization  categories) indexOf: className)  > 0
		ifFalse: [forceOrganization addCategory: className].

	((forceOrganization listAtCategoryNamed: className) indexOf: determinantName) > 0
		ifFalse: [forceOrganization classify: determinantName under: className].

	self forceDeterminant: determinantName.
	self newForceDeterminantList: className.
	self addForceDeterminantInCurrentClass.!

addForceDeterminantInCurrentClass
```

" Agrega el metodo nulo que invoca al super, en todos los SubjectivityCompiledMethods que tenga la clase."

```
	| class listSubjectivityCompiledMethods dictionary aStringSelector codigo |

	class := self selectedClass.
	dictionary := class getMethodDictionary.
	listSubjectivityCompiledMethods :=
		dictionary collect:
			[ :aSubjectivityCompiledMethod |
				(aSubjectivityCompiledMethod class == SubjectivityCompiledMethod)
					ifTrue: [aSubjectivityCompiledMethod].
			].
	listSubjectivityCompiledMethods do:
		[:aSubjectivityCompiledMethod |
			(aSubjectivityCompiledMethod isNil)
			ifFalse:[
				codigo := (aSubjectivityCompiledMethod firstCompiledMethod)
getSource.
				aStringSelector := (self getFullSelectorFrom: codigo).
				aSubjectivityCompiledMethod
								addNullMethodAtForce: forceSelector
								withDeterminant:
forceDeterminantSelector
								andSelector: aStringSelector.
			]
		].!
```

deleteDeterminantInAllSubjectivityCompiledMethods: aDeterminant atForce: aForce
"Borra el determinante de todos los subjectivity compiled methods de la clase"

```
	| SubjectivityCollection |

	SubjectivityCollection := ((self selectedClass) getMethodDictionary)
				select: [ :aSubjectivityCompiledMethod |
					((aSubjectivityCompiledMethod class) ==
SubjectivityCompiledMethod)
				].

	SubjectivityCollection
		do:[ :aSubjectivityCompiledMethod |
			aSubjectivityCompiledMethod remove: aDeterminant atForce: aForce
		].

	SubjectivityCollection
		do: [ :aSubjectivityCompiledMethod |
			aSubjectivityCompiledMethod hasMethods
				ifFalse: [(self selectedClass) removeSelector: selector.
					self newSelectorList: nil
				]
		].!
```

deleteForceDeterminant
"Borra el determinante"

```
	(Dialog confirm: 'Are you certain that you want to remove the Force Determinant ?')
```

```
                ifFalse: [^self].

        " Primero borro en todos los subjectivity compiled methods de la clase "
        self deleteDeterminantInAllSubjectivityCompiledMethods: forceDeterminantSelector
                atForce: forceSelector.

        " Luego borro en el organizer de la Fuerza "
        forceOrganization removeElement: forceDeterminantSelector.
        (forceOrganization listAtCategoryNamed: className)    size > 0
                ifTrue: [forceDeterminantSelector := (forceOrganization listAtCategoryNamed:
className)
                                                                        at: 1
                ]
                ifFalse: [forceDeterminantSelector := nil].

        self newForceDeterminantList: nil.! !
```

!SubjectivityBrowser methodsFor: 'class list'!

className: selection
"Setea el nombre de la clase elegida y actualiza los panes de fuerzas y de logica de fuerzas"

```
        super className: selection.
        self newForceLogicList.
        self forceLogicSelector: nil.! !
```

!SubjectivityBrowser methodsFor: 'text'!

newText
"Restaura el pane de texto"

```
        self resetTheText.
        self text.!
```

text
"Muestra el texto en el pane de acuerdo a las opciones seleccionadas en el browser"

```
        | aMethod aSubjectivityMethod |

        (forceLogicSelector == nil)
                ifFalse: [(self selectedClass == nil)
                                ifTrue: [^super text]

                                ifFalse: [(selector == nil)
                                        ifTrue:  [^SubjectivitySmalltalk
                                                getForceLogicDefinitionFor: (self selectedClass name)
                                                ]
                                        ifFalse: [^SubjectivitySmalltalk
                                                getForceLogicDefinitionFor: (self selectedClass name)
                                                and: selector
                                                ]
                                        ]
                        ].
        "Cuando es methodDefinition veo si es subjetivo o normal"
        (textMode == #methodDefinition)
                ifFalse: [^super text].
```

```
	(selector == nil)
		ifTrue: [^Class sourceCodeTemplate asText].

	aSubjectivityMethod := (self selectedClass) compiledMethodAt: selector.
	((aSubjectivityMethod class) == SubjectivityCompiledMethod)
		ifFalse: [
			(forceButton)
				ifTrue:  [^((self selectedClass) sourceCodeAt: selector) asText
						makeSelectorBoldIn: self classForSelectedMethod
				]
				ifFalse: [^'El metodo tiene implementacion normal' asText].
			]
		ifTrue: [
			(forceButton)
				ifTrue: [^'El metodo tiene implementacion subjetiva' asText]

				ifFalse: [aMethod := aSubjectivityMethod
							compiledMethodAtForce: forceSelector
							withDeterminant: forceDeterminantSelector.
					(aMethod == nil)
						ifTrue: [^Class sourceCodeTemplate asText]
						ifFalse:[
							" Veo si es un metodo definido por el usuario o
uno nulo "
							((aMethod class) == NullCompiledMethod)
								ifTrue: [^Class sourceCodeTemplate
asText]
								ifFalse: [^ (aMethod getSource asText)
									makeSelectorBoldIn:self
									classForSelectedMethod
									]
							]
					]
			].! !
```

!SubjectivityBrowser methodsFor: 'private-selector functions'!

acceptForceLogicDefinition: aText from: aController

```
" Guarda la definicion del bloque de logica de fuerzas para la clase en cuestion. Previamente verifica que
el texto ingresado no contenga errores de compilacion."

	| compiledForceLogicDef |
	" Primero debo asegurarme que el texto ingresado compila correctamente..."
	compiledForceLogicDef := Compiler evaluate: aText notifying: aController logged: false.

	(compiledForceLogicDef == nil)
		ifTrue:   [ " Entonces la compilacion finalizo con error."
				^false.
		]
		ifFalse: [ " Entonces la compilacion fue exitosa. Lo guardo en el ForceLogicDictionary..."
			(selector == nil)
				ifTrue:   [SubjectivitySmalltalk addForceLogicDefinition: aText  in: (self
selectedClass name)]
				ifFalse: [SubjectivitySmalltalk addForceLogicDefinition: aText  in: (self
selectedClass name) and: selector].
```

```
                    ^true.
                    ].!
```

acceptMethod: aText from: aController

```
"Realiza el accept del texto ingresado en el pane de edicion.
 Para eso verifica si se esta ingresando:
        a. La definicion de un metodo subjetivo.
        b. La definicion de un metodo normal (no subjetivo) "

        (self forceButton)
                ifFalse:   [ ^self acceptSubjectivityMethod: aText from: aController]
                ifTrue:    [ ^self acceptNonSubjectivityMethod: aText from: aController].!
```

acceptNonSubjectivityMethod: aText from: aController

```
"Obtiene el nombre del selector, lo busca en el MethodDictionary y verifica si hay un metodo subjetivo.
 En caso de haberlo lo reemplaza, de lo contrario lo da de alta"

        | actualClass actualSelector method response |

        actualClass := self selectedClass.
        actualSelector := self compileText: aText from: aController.
        (actualSelector == nil) ifTrue: [^false].

        (actualClass includesSelector: actualSelector)
                ifTrue: [method := actualClass compiledMethodAt: actualSelector.
                        (method class == CompiledMethod)
                                        ifTrue:   [^super acceptMethod: aText from: aController.]
                                        ifFalse: [response := Dialog confirm: 'Really rewrite the
Subjectivity Method ? \' withCRs initialAnswer: false.
                                                (response)
                                                        ifTrue:   [^super acceptMethod: aText from:
aController]
                                                        ifFalse: [^true].
                                                ]
                        ]
                ifFalse: [^super acceptMethod: aText from: aController].!
```

acceptSubjectivityMethod: aText from: aController

```
"Obtiene el nombre del selector, lo busca en el MethodDictionary y verifica si es un metodo subjetivo. En
caso de serlo, agrega el CompiledMethod del Texto, de lo contrario crea un SubjectivityCompiledMethod"

        | actualClass actualSelector method response  newSubjectivityMethod  |

        actualClass       := self selectedClass.
        actualSelector := self compileText: aText from: aController.
        (actualSelector == nil) ifTrue: [^false].

        (actualClass includesSelector: actualSelector)
        " Busco si existe el metodo en la clase"
                ifTrue: [method := actualClass compiledMethodAt: actualSelector.
                                "Si existe un CompiledMethod, debe
                                pisarlo con un nuevo SubjectivityCompiledMethod"
                                  ((method class) == CompiledMethod)
```

```
                                        ifTrue: [response := Dialog confirm: 'Really rewrite the Compiled
Method ? \' withCRs initialAnswer: false.
                                                (response)
                                                        ifTrue:   [newSubjectivityMethod:= nil]
                                                        ifFalse: [^true].
                                        ]
                                        "El metodo que existe es subjetivo"
                                        ifFalse:[newSubjectivityMethod := method]
                        ]
                " No existe metodo"
                ifFalse:[newSubjectivityMethod := nil].

        self acceptSubjectivityMethod: aText from: aController in: newSubjectivityMethod.

        "Refresco la lista de metodos"
        self newSelectorList: actualSelector.

        ^true.!
```

acceptSubjectivityMethod: aText from: aController in: aSubjectivityCompiledMethod
"Inserta el CompiledMethod resultante de compilar el texto aText en un SubjectivityCompiledMethod. Si aSubjectivityCompiledMethod es nil, entonces de debe crear uno"

```
        | actualCompiledMethod  aFullSelector  newSelector
        selectedClass temporarySubjectivityCompiledMethod |

        temporarySubjectivityCompiledMethod := aSubjectivityCompiledMethod.
        selectedClass := self selectedClass.
        "Compilo el texto aText"
        newSelector := selectedClass compile: aText classified: protocol notifying: aController.
        "Me guardo el CompiledMetod del texto aText compilado"
        actualCompiledMethod:=  (selectedClass compiledMethodAt: newSelector).


        (temporarySubjectivityCompiledMethod  isNil )
                ifTrue: ["Creo el SubjectivityCompiledMethod  con todos los parametros del metodo
(aFullSelector)"
                        aFullSelector := (self getFullSelectorFrom: aText).
                        "Creo el SubjectivityCompiledMethod"
                        temporarySubjectivityCompiledMethod := (SubjectivityCompiledMethod
newForClass: selectedClass withSelector: aFullSelector classified: protocol notifying: aController  )
                ]
                "No hace falta crear el SubjectivityCompiledMethod, pues ya existe,
                la compilacion del texto aText, reemplazo al SubjectivityCompiledMethod
                existente por el CompiledMethod resultante de la compilacion en el
                methodDictionary de la clase, por lo que los vuelvo a cambiar"
                ifFalse:[
                        temporarySubjectivityCompiledMethod
                                        changeInMethodDictionary: newSelector
                                        classified: protocol
                ].
                "Agrego el CompiledMethod resultante de la compilacion de aText al
                SubjectivityCompiledMethod bajo la fuerza y el determinante seleccionados"
                temporarySubjectivityCompiledMethod
                                add: actualCompiledMethod
                                atForce: forceSelector
```

```
                withDeterminant: forceDeterminantSelector.
        ^true.!
```

compileText: aText from: aController
" Compila el texto devolviendo el selector "

```
        |  methodNode |

        methodNode := self getNodeFrom: aText from: aController.
        (methodNode == nil)
                ifTrue:   [^nil]
                ifFalse: [^methodNode selector].!
```

getFullSelectorFrom: aText
" Devuelve el nombre del metodo y de los argumentos a partir del texto aText "

```
        | aStream methodNode |

        " Este metodo ya fue compilado, por eso el controller es nil, de lo contrario nunca hubiera
          llegado a este punto "
        methodNode := self getNodeFrom: aText from: nil.

        aStream:= TextStream on: (Text new).
        (methodNode node block arguments size) == 0
                ifTrue:[ aStream nextPutAll: (methodNode selector asString) ]
                ifFalse:[(methodNode selector keywords) with: (methodNode node block arguments) do:
                                        [:s :arg |
                                        aStream nextPutAll: s; space.
                                        arg printOn: aStream indent:0.
                                        aStream space.
                                        ]
                ].

        ^aStream contents.!
```

getNodeFrom: aText from: aController
" Compila el texto devolviendo un methodNode "

```
        |  methodNode actualClass |

        actualClass := self selectedClass.
        methodNode := actualClass compilerClass new
                                        compile: aText
                                        in: actualClass
                                        notifying: aController
                                        ifFail: [^nil].

        ^methodNode.! !
```

!SubjectivityBrowser methodsFor: 'dolt/accept/explain'!

acceptText: aText from: aController
"Acepta el texto, ya sea de un metodo o de una logica de fuerzas"

```
	(forceLogicSelector == nil)
		ifFalse:[ |accepted |
			accepted := self acceptForceLogicDefinition: aText from: aController.
			aController textHasChanged: accepted not.
			^accepted
		]
		ifTrue: [^super acceptText: aText from: aController].! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

SubjectivityBrowser class
	instanceVariableNames: ''!
```

!SubjectivityBrowser class methodsFor: 'interface specs'!

forceButtonSpec

```
"UIPainter new openOnClass: self andSelector: #forceButtonSpec"

	<resource: #canvas>
	^#(#FullSpec
		#window:
		#(#WindowSpec
			#label: 'Unlabeled Canvas'
			#bounds: #(#Rectangle 105 121 403 206 ) )
		#component:
		#(#SpecCollection
			#collection: #(
				#(#RadioButtonSpec
					#layout: #(#LayoutFrame 0 0 4 0 -19 0.575 22 0 )
					#name: #normalSwitch
					#model: #forceButtonHolder
					#callbacksSpec:
					#(#UIEventCallbackSubSpec
						#requestValueChangeSelector: #changeRequest )
					#label: 'normal'
					#select: true )
				#(#RadioButtonSpec
					#layout: #(#LayoutFrame -19 0.575 4 0 -1 1 24 0 )
					#name: #subjSwitch
					#model: #forceButtonHolder
					#callbacksSpec:
					#(#UIEventCallbackSubSpec
						#requestValueChangeSelector: #changeRequest )
					#label: 'subjective'
					#select: false ) ) ) )!
```

windowSpec

```
"UIPainter new openOnClass: self andSelector: #windowSpec"

	<resource: #canvas>
	^#(#FullSpec
		#window:
		#(#WindowSpec
			#label: 'Subjetive System Browser'
			#min: #(#Point 400 250 )
```

```
        #bounds: #(#Rectangle 21 23 605 425 )
        #isEventDriven: true )
#component:
#(#SpecCollection
        #collection: #(
                #(#SequenceViewSpec
                        #properties:
                        #(#PropertyListDictionary #dragOkSelector
                                #categoryWantToDrag: #dropSelector
                                #categoryDrop: #dragOverSelector
                                #categoryDragOver: #dragStartSelector
                                #doCategoryDrag: #dragEnterSelector
                                #categoryDragEnter: #dragExitSelector
                                #categoryDragLeave: )
                        #layout: #(#LayoutFrame 1 0 1 0 -1 0.25 -1 0.35 )
                        #name: #categoryList
                        #model: #categoryList
                        #callbacksSpec:
                        #(#UIEventCallbackSubSpec
                                #doubleClickSelector: #spawnCategory
                                #requestValueChangeSelector: #changeRequest )
                        #menu: #categoryMenuHolder )
                #(#SequenceViewSpec
                        #properties:
                        #(#PropertyListDictionary #dragOkSelector
                                #classWantToDrag: #dragEnterSelector
                                #classDragEnter: #dropSelector
                                #classDrop: #dragStartSelector
                                #doClassDrag: #dragOverSelector
                                #classDragOver: #dragExitSelector
                                #classDragLeave: )
                        #layout: #(#LayoutFrame 1 0.25 1 0 -1 0.5 -25 0.35 )
                        #name: #classList
                        #model: #classList
                        #callbacksSpec:
                        #(#UIEventCallbackSubSpec
                                #doubleClickSelector: #spawnClassOrHierarchy
                                #requestValueChangeSelector: #changeRequest )
                        #menu: #classMenuHolder )
                #(#SubCanvasSpec
                        #layout: #(#LayoutFrame 0 0.25 -24 0.35 0 0.501562 0 0.35141 )
                        #flags: 0
                        #majorKey: #Browser
                        #minorKey: #metaSpec )
                #(#SequenceViewSpec
                        #properties:
                        #(#PropertyListDictionary #dragOkSelector
                                #protocolWantToDrag: #dragEnterSelector
                                #protocolDragEnter: #dragOverSelector
                                #protocolDragOver: #dragStartSelector
                                #doProtocolDrag: #dropSelector
                                #protocolDrop: #dragExitSelector
                                #protocolDragLeave: )
                        #layout: #(#LayoutFrame 1 0.5 1 0 0 0.753425 -24 0.35 )
                        #name: #protocolList
                        #model: #protocolList
```

```
                #callbacksSpec:
                #(#UIEventCallbackSubSpec
                        #doubleClickSelector: #spawnProtocol
                        #requestValueChangeSelector: #changeRequest )
                #menu: #protocolMenuHolder )
        #(#SequenceViewSpec
                #properties:
                #(#PropertyListDictionary #dragOkSelector
                        #selectorWantToDrag: #dragEnterSelector
                        #selectorDragEnter: #dropSelector
                        #selectorDrop: #dragStartSelector
                        #doSelectorDrag: #dragOverSelector
                        #selectorDragOver: #dragExitSelector
                        #selectorDragLeave: )
                #layout: #(#LayoutFrame 3 0.75 1 0 -3 1 0 0.353579 )
                #name: #selectorList
                #model: #selectorList
                #callbacksSpec:
                #(#UIEventCallbackSubSpec
                        #doubleClickSelector: #spawnMethod
                        #requestValueChangeSelector: #changeRequest )
                #menu: #selectorMenuHolder )
        #(#TextEditorSpec
                #layout: #(#LayoutFrame 0 0.248438 0 0.360087 0 0.995312 0
0.997831 )
                #name: #textValue
                #model: #textValue
                #callbacksSpec:
                #(#UIEventCallbackSubSpec
                        #valueChangeSelector:
                        #textAccepted: )
                #tabable: true
                #menu: #textMenuHolder )
        #(#SubCanvasSpec
                #layout: #(#LayoutFrame 0 0.503125 0 0.299349 0 0.754687 0
0.353579 )
                #flags: 0
                #majorKey: #SubjectivityBrowser
                #minorKey: #forceButtonSpec )
        #(#SequenceViewSpec
                #properties:
                #(#PropertyListDictionary #dragOkSelector
                        #classWantToDrag: #dragEnterSelector
                        #classDragEnter: #dropSelector
                        #classDrop: #dragStartSelector
                        #doClassDrag: #dragOverSelector
                        #classDragOver: #dragExitSelector
                        #classDragLeave: )
                #layout: #(#LayoutFrame -1 0 0 0.410448 0 0.244863 0 0.604478
)
                #name: #forceList
                #model: #forceList
                #callbacksSpec:
                #(#UIEventCallbackSubSpec
                        #doubleClickSelector: #spawnClassOrHierarchy
                        #requestValueChangeSelector: #changeRequest )
```

```
                #menu: #forceMenuHolder )
            #(#SequenceViewSpec
                #properties:
                #(#PropertyListDictionary #dragOkSelector
                    #classWantToDrag: #dragEnterSelector
                    #classDragEnter: #dropSelector
                    #classDrop: #dragStartSelector
                    #doClassDrag: #dragOverSelector
                    #classDragOver: #dragExitSelector
                    #classDragLeave: )
                #layout: #(#LayoutFrame 0 -0.0015625 0 0.607375 0 0.245313 0
0.995662 )
                #name: #forceDeterminantList
                #model: #forceDeterminantList
                #callbacksSpec:
                #(#UIEventCallbackSubSpec
                    #doubleClickSelector: #spawnClassOrHierarchy
                    #requestValueChangeSelector: #changeRequest )
                #menu: #forceDeterminantMenuHolder )
            #(#SequenceViewSpec
                #properties:
                #(#PropertyListDictionary #dragOkSelector
                    #classWantToDrag: #dragEnterSelector
                    #classDragEnter: #dropSelector
                    #classDrop: #dragStartSelector
                    #doClassDrag: #dragOverSelector
                    #classDragOver: #dragExitSelector
                    #classDragLeave: )
                #layout: #(#LayoutFrame 0 0 0 0.348259 0 0.248288 0
0.412935 )
                #name: #forceLogicList
                #flags: 12
                #model: #forceLogicList
                #callbacksSpec:
                #(#UIEventCallbackSubSpec
                    #doubleClickSelector: #spawnClassOrHierarchy
                    #requestValueChangeSelector: #changeRequest )
                #menu: #forceLogicMenuHolder
                #selectionType: #checkMark ) ) ) )! !
```

CLASE SubjectivityCompiledMethod

```
'From VisualWorks® Non-Commercial, Release 3.0 of May 8, 1998 on January 9, 2000 at 8:18:35 am'!

SubjectiveMethod variableSubclass: #SubjectivityCompiledMethod
    instanceVariableNames: 'subjectiveMethods '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'SubjectiveEnvironment'!

SubjectivityCompiledMethod comment:
```

'SubjetivityCompiledMethod es el composite para mantener metodos subjetivos . Tiene una variable de instancia, subjetiveMethods que contiene un arreglo de metodos (Más adelante se puede expandir a un diccionario de metodos indexados por fuerzas).
La clase subjetiva es la que conoce como evaluar las fuerzas y conoce que valor de fuerza hay en el momento de llamar al metodo subjetivo'!

!SubjectivityCompiledMethod methodsFor: 'execution'!

subjectiveMethodFor: anExecutionContext
"Retorna el compiled method que corresponde ejecutar segun la fuerza y determinante que prevalecen en el momento de ejecucion. Para determinar esto, se recurre a la logica de fuerzas que sera evaluada de acuerdo a los valores de anExectionContext."

```
	| anInfluenceForce |

	anInfluenceForce := SubjectivitySmalltalk getInfluenceForceFor: anExecutionContext.
	(anInfluenceForce == nil)
		ifTrue:[ ^self error: 'Falta definicion del bloque de logica de fuerzas en la clase: ',
					(anExecutionContext subjectClass  name),
					', Metodo: ',
					(anExecutionContext selector)
		]
		ifFalse:[^self compiledMethodAtForce: (anInfluenceForce force)
					withDeterminant: (anInfluenceForce
determinant)
		].! !
```

!SubjectivityCompiledMethod methodsFor: 'accesing'!

add: aCompiledMethod   atForce: aForce withDeterminant: aDeterminant
"Agrega el metodo aCompiledMethod en los diccinarios para la fuerza aForce y determinante aDeterminant"

```
	(subjectiveMethods includesKey: aForce)
		ifTrue: [(subjectiveMethods at: aForce) at: aDeterminant put: aCompiledMethod.
		]
		ifFalse: [
			| dictionarysSon |
			dictionarysSon := Dictionary new.
			dictionarysSon at: aDeterminant put: aCompiledMethod.
			subjectiveMethods at: aForce put: dictionarysSon.
		].!
```

addNullMethodAtForce: aForce withDeterminant: aDeterminant andSelector: aStringSelector
" Agrega un compiled method nulo en el determinante aDeterminant de la fuerza aForce, para que invoquen al metodo aStringSelector definido en la superclase "

```
	| code aSelector|

	aSelector:= mclass selectorAtMethod: self ifAbsent:[].
	code := NullCompiledMethod newForClass: mclass
					withSelector: aStringSelector
					classified: '' notifying: nil
```

```
                                        forSubjectiveMethod: self withSelector: aSelector.
        self add: code  atForce: aForce withDeterminant: aDeterminant.!
```

addNullMethodsAtForce: aForce withDeterminants: aListDeterminants andSelector: aSelector
" Agrega compiled methods nulos en todos los determinantes definidos para la fuerza, para que invoquen al metodo definido en la superclase "

```
        aListDeterminants do:   [ :determinant |
                                self addNullMethodAtForce:aForce withDeterminant: determinant
                                        andSelector: aSelector.
                                ].!
```

changeForceName: oldForceName by: newForceName
"Cambia el nombre de la fuerza oldForceName por newForceName."

```
        | methods |
        methods := subjectiveMethods at: oldForceName.
        subjectiveMethods at: newForceName put: methods.
        subjectiveMethods removeKey: oldForceName ifAbsent: [].!
```

compiledMethodAtForce: aForce withDeterminant: aDeterminant
"Devuelve el CompiledMethod que se encuentra definido para la fuerza aForce y determinante aDeterminant"

```
        (subjectiveMethods includesKey: aForce)
                ifFalse: [ ^nil]
                ifTrue:   [((subjectiveMethods at: aForce) includesKey: aDeterminant)
                                ifFalse: [^nil]
                                ifTrue:   [^(subjectiveMethods at: aForce) at: aDeterminant].
                         ].!
```

firstCompiledMethod
" Devuelve el CompiledMethod definido en la primer fuerza para el primer determinante "

```
        | methodsAtPrimerFuerza methodAtPrimerDeterminante |

        methodsAtPrimerFuerza := subjectiveMethods asList first.
        methodAtPrimerDeterminante := (methodsAtPrimerFuerza) asList first.

        ^methodAtPrimerDeterminante.!
```

getInfluenceForceFor: aMethod
"Retorna la Fuerza y el Determinante en las que se encuentra guardado el metodo aMethod"

```
        | force determinant actualDictionary actualDeterminant influenceForce |

        subjectiveMethods do:   [ :d | actualDeterminant := d keyAtValue: aMethod ifAbsent: [nil].
                                   (actualDeterminant == nil) ifFalse: [determinant := actualDeterminant.
                                                                        actualDictionary := d
                                                                        ].
                                ].
        force := subjectiveMethods keyAtValue: actualDictionary ifAbsent: [nil].
        influenceForce := InfluenceForce new.
        influenceForce force: force.
        influenceForce determinant: determinant.
        ^influenceForce.!
```

hasMethods
"Indica si el SubjectivityCompiledMethod tiene metodos definidos"

```
	| size |
	size := 0.
	subjectiveMethods do: [ :l | size := size + l size].
	^ (size > 0).!
```

includes: aMethod
"Indica si el SubjectivityCompiledMethod tiene definido el metodo aMethod"

```
	subjectiveMethods do:
		[ :aCompiledMethodDictionary |
			(aCompiledMethodDictionary includes: aMethod)
				ifTrue: [^true]
		].
	^false.!
```

remove: aDeterminant atForce: aForce
"Borra el elemento cuya clave es aDeterminant del diccionario de determinantes"

```
	(subjectiveMethods includesKey: aForce)
		ifTrue:[((subjectiveMethods at: aForce) includesKey: aDeterminant)
				ifTrue: [(subjectiveMethods at: aForce) removeKey: aDeterminant]
		].! !
```

!SubjectivityCompiledMethod methodsFor: 'private'!

initializeInClass: aClass withFullSelector: aFullSelector withSelector: aSelector forMethod: aCompiledMethod classified: aProtocol
" Inicializa el diccionario de fuerzas y determinantes, poniendo en cada determinante definido un compiled method para el mensaje aSelector, para que invoque al mensaje de la superclase. Inicializa los bytecodes y lo coloca en el diccionario de metodos de manera correcta. "

```
	| listForces listDeterminants |

	mclass := aClass.
	subjectiveMethods:= Dictionary new.

	listForces := SubjectivitySmalltalk getForceNameList.
	listForces do:	[ :force |
				listDeterminants := SubjectivitySmalltalk getDeterminantListForForce:
force 					intoClass: mclass.
				self addNullMethodsAtForce: force withDeterminants: listDeterminants
					andSelector: aFullSelector.
			].
	self bytecodesFor: aCompiledMethod.
	^ self changeInMethodDictionary: aSelector classified: aProtocol.! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

SubjectivityCompiledMethod class
	instanceVariableNames: "!
```

!SubjectivityCompiledMethod class methodsFor: 'templates'!

executionTemplate: aStringSelector
"Retorna el texto del codigo que corresponde a un SubjectivityCompiledMethod. Este codigo obtiene la fuerza influyente sobre el objeto y deriva la ejecucion al CompiledMetod correspondiente"

```
^Text fromString: (aStringSelector,'
        | aCompiledMethod methodArguments anExecutionContext |

        methodArguments := Array new: (thisContext method numArgs).
        1 to: (thisContext method numArgs) do:
       [:i | methodArguments at: i put: (thisContext localAt: i)].

        anExecutionContext:= ExecutionContext new.
        anExecutionContext subjectClass: (thisContext method mclass).
        anExecutionContext subject: self.
        anExecutionContext sender: ( (thisContext sender) receiver).
        anExecutionContext arguments: methodArguments.
        anExecutionContext selector: (thisContext selector).

        aCompiledMethod   :=  (thisContext method) subjectiveMethodFor: anExecutionContext.
        ^aCompiledMethod valueWithReceiver: (thisContext receiver)
                        arguments: methodArguments.'
).! !
```

CLASE SubjectivityDebugger

'From VisualWorks® Non-Commercial, Release 3.0 of May 8, 1998 on January 9, 2000 at 8:18:37 am'!

Debugger subclass: #SubjectivityDebugger
instanceVariableNames: 'forceValue determinantValue subjectivityMethod influenceForce isForceLogic forceLogicKey '
classVariableNames: ''
poolDictionaries: ''
category: 'SubjectiveEnvironment'!
SubjectivityDebugger comment:
'SubjectivityDebugger es un Debugger para el ambiente subjetivo, proporciona al programador los mecanismos necesarios realizar el seguimiento de un proceso suspendido, manejando metodos normal o subjetivo, en cuyo caso agregando la fuerza y el determinante en la que fue definido el metodo.

Instance Variables:

forceValue	<ClassOfVariable>	description of variable"s function
determinantValue	<ClassOfVariable>	description of variable"s function
subjectivityMethod	<ClassOfVariable>	description of variable"s function
influenceForce	<ClassOfVariable>	description of variable"s function
isForceLogic	<ClassOfVariable>	description of variable"s function
forceLogicKey	<ClassOfVariable>	description of variable"s function'!

!SubjectivityDebugger methodsFor: 'aspects'!

determinantValue
"This method was generated by UIDefiner. Any edits made here may be lost whenever methods are automatically defined. The initialization provided below may have been preempted by an initialize method.
Devuelve el valor del determinante del metodo subjetivo."

```
^determinantValue isNil
        ifTrue:
                [determinantValue := String new asValue]
        ifFalse:
                [determinantValue]!
```

forceValue
"This method was generated by UIDefiner. Any edits made here may be lost whenever methods are automatically defined. The initialization provided below may have been preempted by an initialize method.
Devuelve el valor de la fuerza del metodo subjetivo."

```
^forceValue isNil
        ifTrue:
                [forceValue := String new asValue]
        ifFalse:
                [forceValue]! !
```

!SubjectivityDebugger methodsFor: 'context list'!

context: aContext
" Muestro el contexto y veo si es Subjetivo, normal o Logica de Fuerzas. En caso de ser Subjetivo, obtengo la fuerza y el determinante del metodo dentro del contexto aContext. En caso de ser Logica de Fuerzas lo marco como tal"

```
super context: aContext.

(self isForceLogicBlock: aContext)
        ifTrue:   [subjectivityMethod := nil]
        ifFalse: [self findSubjectivityMethodIn: aContext].

self setInfluenceForceFor: aContext.! !
```

!SubjectivityDebugger methodsFor: 'subjectivity'!

findSubjectivityMethodIn: aContext
"Busca el SubjectivityCompiledMethod que contiene al metodo del contexto aContext"

```
| method methodDict |

subjectivityMethod := nil.
(aContext == nil)
        ifFalse: [
                method := aContext method.
```

```
                ((method class == CompiledMethod)
                        or: [ method class == NullCompiledMethod])
                ifTrue: [ methodDict  := (aContext mclass) getMethodDictionary.
                                methodDict do: [ :m |
                                                        (((m class)  ==
SubjectivityCompiledMethod) and: [m includes: method])
                                                                ifTrue: [subjectivityMethod := m]
                                                ].
                ]
                ifFalse: [subjectivityMethod:= method]
        ].!
```

isForceLogicBlock: aContext
"Dice si el codigo del metodo del contexto aContext es un bloque de Logica de Fuerzas. Ademas guarda en la variable de instancia forceLogicKey el nombre del metodo o de la Clase en el que se definio la Logica de Fuerzas"

```
        | text |

        (aContext == nil)
                ifTrue: [^false].

        text := aContext method getSource asText.

        forceLogicKey := SubjectivitySmalltalk getSelectorFor: text in: (aContext mclass name).
        (forceLogicKey == nil)
                ifTrue:   [isForceLogic := false]
                ifFalse: [isForceLogic := true].

        ^isForceLogic!
```

setInfluenceForceFor: aContext
"Setea la Fuerza y el Determinante correspondientes (que influenciaron al objeto) al metodo en aContext dentro del SubjectivityCompiledMethod subjectivityMethod"

```
        (subjectivityMethod == nil)
                ifTrue: [(self isForceLogicBlock: aContext)
                                ifFalse: [forceValue setValue: String new.
                                                determinantValue setValue: String new.
                                                influenceForce := InfluenceForce new.
                                ]
                                ifTrue:   [forceValue setValue: 'LOGIC'.
                                                determinantValue setValue: forceLogicKey.
                                                influenceForce := InfluenceForce new.
                                ]
                ]
                ifFalse: [ |method|
                                  method:= aContext method.
                                 (( (method class) == CompiledMethod ) or:[method class ==
NullCompiledMethod ])
                                        ifTrue:[  influenceForce := subjectivityMethod
getInfluenceForceFor: method.
                                                forceValue setValue: influenceForce force.
```

```
                                        determinantValue setValue: influenceForce determinant.
                                ]
        ].

        forceValue changed.
        determinantValue changed.! !
```

!SubjectivityDebugger methodsFor: 'initialize-release'!

initialize
"Inicializa el SubjectivityDebugger"

```
    super initialize.
    forceValue := ValueHolder new.
    determinantValue := ValueHolder new.! !
```

!SubjectivityDebugger methodsFor: 'private-selector functions'!

acceptForceLogicDefinition: aText from: aController
" Guarda la definicion del bloque de logica de fuerzas para la clase en cuestion. Previamente verifica que el texto ingresado no contenga errores de compilacion."

```
    | compiledForceLogicDef newMethod newContext classOfMethod |

    " Primero debo asegurarme que el texto ingresado compila correctamente..."
    classOfMethod := context mclass.
    compiledForceLogicDef := Compiler evaluate: aText notifying: aController logged: false.

    (compiledForceLogicDef == nil)
        ifTrue:     [ " Entonces la compilacion finalizo con error."
                      ^false.
                    ].

    compiledForceLogicDef method outerMethod sourcePointer: (aText asString).
    compiledForceLogicDef method outerMethod mclass: classOfMethod.

    " Entonces la compilacion fue exitosa. Lo guardo en el ForceLogicDictionary..."
    SubjectivitySmalltalk addForceLogicDefinition: aText in: (self selectedClass name) and:
forceLogicKey.

    Cursor execute showWhile:
        [newMethod :=  compiledForceLogicDef method.
        sourceCode := aText string.
        newContext := context copy.
        newContext method: newMethod.
        processHandle cutbackTo: newContext.
        newContext := newContext resizedWith: newMethod.
        self resetContext: newContext.
        processHandle interrupted: true].

    ^true.!
```

acceptSubjectivityMethod: aText from: aController in: aSubjectivityCompiledMethod
"Inserta un Compiled Method en el SubjectivityCompiledMethod correcto"

```
	| selectedClass  newCompiledMethod  newSelector actualProtocol newMethod newContext
newText|

	"Si la logica de Fuerza es nula, entonces esta tratando de modificar el codigo interno del
	 SubjectityCompiledMethod lo cual no esta permitido"

	(forceValue value = '' )
		ifTrue:[
			Dialog warn:
				'This is a Method for a SubjectivityCompiledMethod.
				 You cannot modify it'.
			^true.
		].

	newText := aText.
	selectedClass := self selectedClass.
	newSelector := selectedClass compile: newText
				classified: (ClassOrganizer defaultProtocol)
				notifying: aController.
	actualProtocol := ((selectedClass organization) categoryOfElement: newSelector).

	( (aSubjectivityCompiledMethod
			compiledMethodAtForce: (forceValue value)
			withDeterminant: (determinantValue value) class) == CompiledMethod)
	ifTrue:[
			newCompiledMethod:=  (selectedClass compiledMethodAt: newSelector).
			aSubjectivityCompiledMethod
				add: newCompiledMethod
				atForce: (forceValue value)
				withDeterminant: (determinantValue value).
	]
	ifFalse:[
			"No se puede cambiar un NullCompiledMethod en el Debbugger, reestablezco
los			valores anteriores"
			Dialog warn:
				'This is a NullMethod for a SubjectivityCompiledMethod.
				You cannot modify it'.
			newCompiledMethod :=
				aSubjectivityCompiledMethod
					compiledMethodAtForce: (forceValue value)
					withDeterminant: (determinantValue value) .
			newText := newCompiledMethod getSource asText.
	].
	aSubjectivityCompiledMethod
		changeInMethodDictionary: newSelector
		classified: actualProtocol.

	Cursor execute showWhile:
		[newMethod := newCompiledMethod.
		sourceCode := newText string.
		newContext := self homeContext.
		processHandle cutbackTo: newContext.
		newContext := newContext resizedWith: newMethod.
		self resetContext: newContext.
		processHandle interrupted: true].
	^ true! !
```

!SubjectivityDebugger methodsFor: 'doIt/accept/explain'!

acceptText: aText from: aController

```
"Recompile the method of the selected context."

	" Si es la Logica de Fuerzas ==> la compilo y guardo el texto en ForceLogicDictionary "
	(isForceLogic)
		ifTrue: [^self acceptForceLogicDefinition: aText from: aController].

	" Si es un CompiledMethod ==> no hago nada nuevo "
	(subjectivityMethod == nil)
		ifTrue: [^super acceptText: aText from: aController].
	" Si es un SubjectivityCompiledMethod, compilo el codigo y lo guardo bajo la fuerza y el
	determinante "
	^self acceptSubjectivityMethod: aText from: aController in: subjectivityMethod.! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

SubjectivityDebugger class
	instanceVariableNames: ''!
```

!SubjectivityDebugger class methodsFor: 'interface specs'!

windowSpec

```
"UIPainter new openOnClass: self andSelector: #windowSpec"

	<resource: #canvas>
	^#(#FullSpec
		#window:
		#(#WindowSpec
			#label: ' '
			#min: #(#Point 300 230 )
			#bounds: #(#Rectangle 262 128 562 358 ) )
		#component:
		#(#SpecCollection
			#collection: #(
				#(#SequenceViewSpec
					#layout: #(#LayoutFrame 1 0 1 0 -1 1 -1 0.2 )
					#name: #contextListHolder
					#model: #contextListHolder
					#callbacksSpec:
					#(#UIEventCallbackSubSpec
						#requestValueChangeSelector: #changeRequest )
					#menu: #contextListMenuHolder )
				#(#ActionButtonSpec
					#layout: #(#LayoutFrame 2 0 0 0.2 53 0 25 0.2 )
					#name: #step
					#model: #step
					#label: 'step' )
				#(#TextEditorSpec
					#layout: #(#LayoutFrame 1 0 27 0.2 -1 1 -1 0.8 )
					#name: #textValue
					#model: #textValue
					#callbacksSpec:
					#(#UIEventCallbackSubSpec
```

```
                                        #valueChangeSelector:
                                        #textAccepted: )
                                #tabable: true
                                #menu: #textMenuHolder )
                        #(#SubCanvasSpec
                                #layout: #(#LayoutFrame 0 0 0 0.8 0 0.5 0 1 )
                                #name: #receiverInspector
                                #flags: 0
                                #majorKey: #Inspector
                                #minorKey: #windowSpec
                                #clientKey: #receiverInspector )
                        #(#SubCanvasSpec
                                #layout: #(#LayoutFrame 0 0.5 0 0.8 0 1 0 1 )
                                #name: #contextInspector
                                #flags: 0
                                #majorKey: #Inspector
                                #minorKey: #windowSpec
                                #clientKey: #contextInspector )
                        #(#ActionButtonSpec
                                #layout: #(#LayoutFrame 54 0 0 0.2 106 0 25 0.2 )
                                #name: #send
                                #model: #send
                                #label: 'send' )
                        #(#InputFieldSpec
                                #layout: #(#LayoutFrame 0 0.503125 0 0.201735 0 0.659375 0
0.255965 )
                                #name: #forceValue
                                #model: #forceValue
                                #tabable: false
                                #isReadOnly: true )
                        #(#LabelSpec
                                #layout: #(#LayoutOrigin 0 0.678125 0 0.206074 )
                                #label: 'Determinant :' )
                        #(#InputFieldSpec
                                #layout: #(#LayoutFrame 0 0.820312 0 0.201735 0 0.976562 0
0.255965 )
                                #name: #determinantValue
                                #model: #determinantValue )
                        #(#LabelSpec
                                #layout: #(#LayoutFrame 0 0.417187 0 0.208243 0 0.498437 0
0.253796 )
                                #label: 'Force :' ) ) ) )! !
```

CLASE SubiectivitvVisualLauncher

```
'From VisualWorks® Non-Commercial, Release 3.0 of May 8, 1998 on January 9, 2000 at 8:18:39 am'!

VisualLauncher subclass: #SubjectivityVisualLauncher
        instanceVariableNames: ''
        classVariableNames: ''
```

poolDictionaries: ''
category: 'SubjectiveEnvironment'!

SubjectivityVisualLauncher comment:
'SubjectivityVisualLauncher es un VisualLauncher, que permite crear launcher"s subjetivos, para manejar las caracteristicas del ambiente subjetivo:browsers subjetivos, opciones de menu propias, herramientas de administracion de las fuerzas, etc. '!

!SubjectivityVisualLauncher methodsFor: 'actions'!

adminForces
"Abre el administrador de Fuerzas"

ForceAdministrator open.!

browseAllClasses
"Open new Subjectivity System Browser"

self openApplicationForClassNamed: #SubjectivityBrowser!

browseAllParcels
"Open new Parcel Browser"

self error: 'Not implemented in this version' .!

browseClassNamed
"Get user to pick a class name before opening a browser"

self openApplicationForClassNamed: #SubjectivityBrowser withSelector: #newPickClass!

browseGlobal
"Browse references to a global variable (either in Smalltalk or Undeclared). Try to track down references that go through the global's name as well as direct references via its association, e.g. Smalltalk at: UnixProcess"

self openApplicationForClassNamed: #SubjectivityBrowser withSelector: #browseGlobal!

browseImplementorsOf
"Muestra los metodos que implementan el mensaje solicitado"

self openApplicationForClassNamed: #SubjectivityBrowser withSelector: #promptThenBrowseImplementors!

browseMethodsInProtocol
"Muestra los metodos que se encuentran definidos en el protocolo solicitado"

self openApplicationForClassNamed: #SubjectivityBrowser
withSelector: #promptThenBrowseMethodsInProtocol!

browseSendersOf
"Muestra los emisores del mensaje solicitado"

self openApplicationForClassNamed: #SubjectivityBrowser withSelector: #promptThenBrowseCalls!

visualWorksLauncher
"Abre el Launcher del ambiente subjetivo"

```
	 self class open! !
```

!SubjectivityVisualLauncher methodsFor: 'relaunch'!

winNewLauncher
"Create a new subjective launcher."

```
	self  closeAndUnschedule.
	self class open! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!
```

```
SubjectivityVisualLauncher class
	instanceVariableNames: ''!
```

!SubjectivityVisualLauncher class methodsFor: 'accessing'!

title
"Return a title for the launcher which includes the image name"

```
	^'Subjective VisualWorks Non-Commercial ' , ObjectMemory imagePrefix! !
```

!SubjectivityVisualLauncher class methodsFor: 'resources'!

launcherToolBar
"MenuEditor new openOnClass: self andSelector: #launcherToolBar"

```
	<resource: #menu>
	^#(#Menu #(
				#(#MenuItem
					#rawLabel: 'File Browser'
					#nameKey: #filesButton
					#value: #openFileList
					#labelImage: #(#ResourceRetriever nil #fileListIcon ) )
				#(#MenuItem
					#rawLabel: 'Workspace'
					#value: #toolsNewWorkspace
					#labelImage: #(#ResourceRetriever nil #workspaceIcon ) )
				#(#MenuItem
					#rawLabel: 'Browse Classes'
					#nameKey: #classesButton
					#value: #browseAllClasses
					#labelImage: #(#ResourceRetriever nil #allClassesIcon ) )
				#(#MenuItem
					#rawLabel: 'Browse Parcels'
					#nameKey: #parcelsButton
					#value: #browseAllParcels
					#labelImage: #(#ResourceRetriever nil #allParcelsIcon ) )
				#(#MenuItem
					#rawLabel: 'Admin Forces'
					#nameKey: #afButton
					#value: #adminForces ) ) #(5 ) nil ) decodeAsLiteralArray!
```

menuBar
"MenuEditor new openOnClass: self andSelector: #menuBar"

```
        <resource: #menu>
        ^#(#Menu #(
                        #(#MenuItem
                                #rawLabel: '&File'
                                #nameKey: #file
                                #submenu: #(#Menu #(
                                                #(#MenuItem
                                                        #rawLabel: '&Save As...'
                                                        #value: #imageSaveAs )
                                                #(#MenuItem
                                                        #rawLabel: '&Perm Save As...'
                                                        #value: #filePermSaveAs )
                                                #(#MenuItem
                                                        #rawLabel: 'Perm &Undo As...'
                                                        #value: #filePermUndoAs )
                                                #(#MenuItem
                                                        #rawLabel: '&Collect Garbage'
                                                        #value: #collectGarbage )
                                                #(#MenuItem
                                                        #rawLabel: 'Collect All &Garbage'
                                                        #value: #collectAllGarbage )
                                                #(#MenuItem
                                                        #rawLabel: 'Se&ttings'
                                                        #value: #visualWorksSettings )
                                                #(#MenuItem
                                                        #rawLabel: 'E&xit VisualWorks...'
                                                        #value: #visualWorksExit ) ) #(3 2 1 1 ) nil ) )
                        #(#MenuItem
                                #rawLabel: '&Browse'
                                #nameKey: #browse
                                #submenu: #(#Menu #(
                                                #(#MenuItem
                                                        #rawLabel: '&All Classes'
                                                        #value: #browseAllClasses
                                                        #labelImage: #(#ResourceRetriever nil
#allClassesIcon ) )
                                                #(#MenuItem
                                                        #rawLabel: 'Class &Named...'
                                                        #value: #browseClassNamed )
                                                #(#MenuItem
                                                        #rawLabel: 'All Parcels'
                                                        #value: #browseAllParcels
                                                        #labelImage: #(#ResourceRetriever nil
#allParcelsIcon ) )
                                                #(#MenuItem
                                                        #rawLabel: 'Admin &Forces'
                                                        #value: #adminForces )
                                                #(#MenuItem
                                                        #rawLabel: 'References &To...'
                                                        #value: #browseSendersOf )
                                                #(#MenuItem
                                                        #rawLabel: '&Implementors Of...'
                                                        #value: #browseImplementorsOf )
```

```
                                        #(#MenuItem
                                                #rawLabel: 'Refs to Global...'
                                                #value: #browseGlobal )
                                        #(#MenuItem
                                                #rawLabel: 'Inspect...'
                                                #value: #browseInspect ) ) #(4 3 1 ) nil ) )
                        #(#MenuItem
                                #rawLabel: '&Tools'
                                #nameKey: #tools
                                #submenu: #(#Menu #(
                                        #(#MenuItem
                                                #rawLabel: '&File List'
                                                #value: #openFileList
                                                #labelImage: #(#ResourceRetriever nil
#fileListIcon ) )
                                        #(#MenuItem
                                                #rawLabel: 'File &Editor...'
                                                #value: #openFileEditor )
                                        #(#MenuItem
                                                #rawLabel: '&Workspace'
                                                #value: #toolsNewWorkspace
                                                #labelImage: #(#ResourceRetriever nil
#workspaceIcon ) )
                                        #(#MenuItem
                                                #rawLabel: '&Advanced'
                                                #nameKey: #advanced )
                                        #(#MenuItem
                                                #rawLabel: '&DLL and C Connect'
                                                #nameKey: #dllcc
                                                #value: #openExternalFinder
                                                #labelImage: #(#ResourceRetriever nil
#extFinderIcon ) )
                                        #(#MenuItem
                                                #rawLabel: 'System &Transcript'
                                                #nameKey: #transcript
                                                #value: #toggleSystemTranscript
                                                #indication: true ) ) #(3 2 1 ) nil ) )
                        #(#MenuItem
                                #rawLabel: '&Changes'
                                #nameKey: #changes
                                #submenu: #(#Menu #(
                                        #(#MenuItem
                                                #rawLabel: 'Open Change &List'
                                                #value: #changesOpenChangeList )
                                        #(#MenuItem
                                                #rawLabel: 'Open Change Sets'
                                                #value: #changesOpenChangeSets )
                                        #(#MenuItem
                                                #rawLabel: 'Condense Changes'
                                                #value: #changesFileCondense ) ) #(2 1 ) nil ) )
                        #(#MenuItem
                                #rawLabel: '&Database'
                                #nameKey: #database
                                #submenu: #(#Menu #(
                                        #(#MenuItem
                                                #rawLabel: 'Ad Hoc &SQL'
```

```
                                        #nameKey: #adHoc
                                        #value: #openAdHocQuery )
                                    #(#MenuItem
                                        #rawLabel: 'Data &Modeler'
                                        #nameKey: #dataModeler
                                        #value: #openDataModelBrowser
                                        #labelImage: #(#ResourceRetriever nil
#dbToolIcon ) )
                                    #(#MenuItem
                                        #rawLabel: 'Canvas &Composer'
                                        #nameKey: #canvasComposer
                                        #value: #openCanvasComposer )
                                    #(#MenuItem
                                        #rawLabel: 'New Data &Form...'
                                        #nameKey: #dataForm
                                        #value: #newDataForm )
                                    #(#MenuItem
                                        #rawLabel: 'New Database &Application...'
                                        #nameKey: #dataBaseAp
                                        #value: #newDataMain ) ) #(1 4 ) nil ) )
                        #(#MenuItem
                            #rawLabel: '&Window'
                            #nameKey: #window
                            #submenu: #(#Menu #(
                                    #(#MenuItem
                                        #rawLabel: 'Re&fresh All'
                                        #value: #winRefreshAll )
                                    #(#MenuItem
                                        #rawLabel: '&Collapse All'
                                        #value: #winCollapseAll )
                                    #(#MenuItem
                                        #rawLabel: 'Re&store All'
                                        #value: #winRestoreAll )
                                    #(#MenuItem
                                        #rawLabel: '&Windows'
                                        #nameKey: #windowsMenu )
                                    #(#MenuItem
                                        #rawLabel: 'New Launcher'
                                        #value: #winNewLauncher ) ) #(3 1 1 ) nil ) )
                        #(#MenuItem
                            #rawLabel: '&Help'
                            #nameKey: #help
                            #submenu: #(#Menu #(
                                    #(#MenuItem
                                        #rawLabel: 'Open Online &Documentation'
                                        #nameKey: #helpBrowser
                                        #value: #openHelpBrowser
                                        #labelImage: #(#ResourceRetriever nil
#helpIcon ) )
                                    #(#MenuItem
                                        #rawLabel: '&Quick Start Guides...'
                                        #nameKey: #quickStartGuides
                                        #value: #openGuidingDialog )
                                    #(#MenuItem
                                        #rawLabel: 'About &VisualWorks Non-
Commercial...'
```

#value: #helpAbout)) #(2 1) nil))) #(7) nil)
decodeAsLiteralArray! !