Implicaciones de Eiffel en el Diseño Orientado a Objetos (fragmento)

Francisco J. García Peñalvo
Yania Crespo González-Carvajal
en RPP no. 42 (Julio/Agosto) 1998 pág:45-53

...

Estructura de una clase Eiffel

Una clase Eiffel puede dividirse de forma general en: cabecera, recursos e invariantes. Esto no es más que una manera de agrupar las construcciones lingüísticas correspondientes a la clase con fines didácticos pues una clase Eiffel es un elemento indivisible, al contrario de C++ donde la especificación de la clase puede dividirse en declaraciones y definiciones (implementación), llegando incluso a disgregarse en un fichero cabecera que contiene la declaración (.h) y en un fichero que contiene las definiciones de los métodos de la clase (.cpp).

En la cabecera de una clase Eiffel se inscriben sus declaraciones generales: nombre, especificaciones de indexación, cláusula de herencia, cláusula de creación, etc. La segunda parte, que hemos nombrado recursos, constituye una declaración y definición (si procede) de los recursos propios de la clase. La unificación en un mismo concepto de atributos y métodos, donde los métodos pueden ser de tipo función (retornan un resultado) o de tipo rutina, es lo que se denomina en Eiffel recursos (**features**) de una clase. En la última parte, como su nombre lo indica, se especifican las expresiones lógicas que constituyen los invariantes de la clase, es decir, las propiedades que un objeto construido a partir de dicha clase debe conservar a lo largo de todo su ciclo de vida.

Como primer elemento de la cabecera de una clase se tiene la cláusula de indexación, que se identifica sintácticamente por la palabra clave **index**, en la que se agrupan un conjunto de pares etiqueta-valor. Las etiquetas más habituales que pueden ser incluidas son aquellas que indican los sinónimos (**synonyms**), palabras claves (**keywords**) y dominios de aplicación (**domains**), etc. de la clase. La especificación de esta cláusula es completamente opcional, al igual que los pares etiqueta-valor. Ahora bien, la existencia de esta cláusula como parte de la sintaxis de la estructura de una clase es un hecho importante. La información que aparece en la cláusula de indexación está presente por defecto en cualquiera de las formas estándar de recuperación de información de clases Eiffel precompiladas (forma *short*, forma *flat* [2, 3]). Además, constituye un punto de partida para las herramientas de localización de clases en repositorios o bibliotecas.

A continuación de la cláusula de indexación (que al ser opcional puede no aparecer) se tiene el nombre de la clase, por ejemplo MATRIZ, que puede venir acompañado de la indicación de si esta clase es genérica y, por consiguiente, de la declaración de los parámetros genéricos formales con sus restricciones, MATRIZ[T→NUMERIC]. Esto indica que la clase MATRIZ es una clase genérica con un único parámetro genérico formal identificado por T. La restricción que viene señalada por →, indica que el tipo real con el que se instancie el genérico formal T tiene que conformar con el tipo NUMERIC. La conformancia de tipos en Eiffel está basada principalmente en la herencia de clases (aunque en realidad la regla es más compleja debido a la genericidad [1]). NUMERIC es una clase de la jerarquía de clases predefinida de Eiffel de la cual descienden algunos tipos básicos como INTEGER, REAL, etc. y que define básicamente a un elemento que tiene las operaciones aritméticas (suma, multiplicación, etc.). El programador puede crear un elemento numérico propio heredando de NUMERIC y definiendo dichas operaciones como puede ser el caso de definir una clase COMPLEJO que describa los elementos del campo de los números complejos.

En la cláusula de herencia, que se identifica sintácticamente por la palabra clave **inherit**, se listan los ancestros directos de la clase. Para cada padre pueden incluirse diferentes subcláusulas que especifican,

si es el caso, qué recursos heredados se renombran y cómo (cláusula **rename**), qué recursos heredados se redefinen (cláusula **redefine**), qué métodos pierden su definición, con lo cual se consigue que la clase pase a ser una clase abstracta porque no se aporta definición para dicho recurso (cláusula **undefine**) y qué recursos cambiarán su exportación y cómo (cláusula **export**), entre otras.

Entre los métodos que tiene una clase se puede especificar cuál o cuales de ellos constituyen métodos de creación. Esto se hace a través de la cláusula de creación, identificada por la palabra clave **creation**, que es una lista de uno o más métodos. Los métodos que se declaran en dicha cláusula son los que pueden aparecer como parte de una instrucción de creación de objetos de la clase. Las acciones más comunes que se realizan en estos métodos pasan por la inicialización de los atributos del objeto de modo que desde su nacimiento satisfagan los invariantes de la clase. Un método puede exportarse como método de creación y como otro más de los recursos de la clase. Si este es el caso, además de poderse utilizar como parte de una instrucción de creación, puede ser invocado en cualquier momento de la vida del objeto (como un método más). En cambio, también es posible especificar que un método se utilice sólo con fines de creación. Esto se hace incluyéndolo en la cláusula de creación y, por otra parte indicando que no se exporte como recurso (en la parte correspondiente a los recursos).

El otro gran grupo, en el que hemos dividido la estructura de una clase Eiffel, corresponde a los recursos. La palabra clave **feature** identifica el comienzo de una sección en la que se definirán uno o más recursos de la clase. Cada grupo de recursos (marcado por la palabra **feature**) puede ir acompañado de una especificación de exportación. Los recursos pueden exportarse a todas, a ninguna o a una o más clases. Si no se especifica nada, por defecto, se asume la exportación a {ANY}, es decir a todas las clases. Para no exportar a ninguna clase se declara {NONE}. Para exportar los recursos a una o varias clases se lista entre llaves el nombre de estas.

Como parte de las especificaciones de recursos, se definen atributos variables o constantes y métodos rutinas o funciones. La definición de un atributo variable consiste en especificar su nombre y su tipo. La definición de un atributo constante, en indicar nombre, tipo y valor. La especificación de un método empieza por indicar si es diferido o no (abstracto o virtual puro en términos de C++) y declarar su signatura (nombre, nombre y tipo de los parámetros y tipo del resultado para el caso de las funciones). Como parte de la declaración del método, independientemente de si es o no diferido, se pueden especificar sus precondiciones y postcondiciones. Si el método no es diferido se acompaña de su definición, es decir, la secuencia de instrucciones que constituyen su implementación.

En la parte correspondiente a los recursos también aparecerá la nueva definición que la clase actual da para aquellos recursos heredados que se redefinirán, si así se especifica en la subcláusula **redefine** de la cláusula **inherit**. Una redefinición puede ser un cambio del tipo de un atributo, de un parámetro de un método o del resultado de una función. El nuevo tipo deberá conformar con el anterior (redefinición covariante). También, una función sin parámetros puede redefinirse, cuando se hereda, como un atributo (pero no viceversa). Por ejemplo, si se tiene una clase PERSONA con una función que retorna la edad, calculada a partir de la fecha de nacimiento, en una clase que heredera puede redefinirse la función edad como un atributo. Dicha redefinición puede ir acompañada de incluir otro método en la clase que incremente la edad el día del cumpleaños de la persona. Por último, en general, un método puede redefinirse proporcionando otra implementación y en consecuencia, las precondiciones y postcondiciones podrán cambiarse también, pero no libremente [1,2]. Ni los atributos constantes, ni los recursos declarados como **frozen** pueden redefinirse.

La que se ha clasificado como última parte de la estructura de una clase, es la que corresponde a la definición de sus invariantes. Esta sección se identifica sintácticamente con la palabra clave **invariant**. La especificación de los invariantes no es más que declarar un conjunto de expresiones lógicas en las que están involucrados los recursos de la clase. Con la herencia pueden añadirse nuevos invariantes para la nueva clase, pero tienen que respetarse los que se indican en los ancestros. En el listado 1 se presenta un ejemplo de una clase Eiffel.

Listado 1 Ejemplo de la clase Matriz[T → NUMERIC] en Eiffel.

Texto de la clase MATRIZ [T → NUMERIC]

```
indexing
        description: "Matrices genéricas con parámetro restringido
                      a un tipo al que se le pueden aplicar
                     operaciones aritméticas";
        domains: "métodos algebraicos, resolución de ecuaciones"

class MATRIZ[T->NUMERIC]
creation
  creaMatriz
feature {NONE} – Representación interna
  elementos : ARRAY[ARRAY[T]];
feature  -- Acceso
  --dimensiones
  filas : INTEGER;
  columnas : INTEGER;
  -- Creación
  creaMatriz(n, m : INTEGER) is
      -- Crea una matriz con dimensiones n x m con todos los
      -- elementos inicializados con el valor por defecto de T
  require
    positivo: n > 0 and m > 0
  local
     i : INTEGER;
     aux : ARRAY[T];
  do
    filas := n;
    columnas := m;
    !!elementos.make(1,columnas);
    from i:=1
    until i > columnas
    loop
      !!aux.make(1, filas);
      elementos.put(aux,i);
      i := i + 1;
    end;
  ensure
     dimensiones : filas = n and columnas = m
  end;

-- Trabajo con los elementos de la MATRIZ
  celda(i,j : INTEGER) : T is
  -- Retorna el elemento en la posición (i,j) de la matriz
  require
     rango: 0 < i and  i <= filas and 0 < j and j <= columnas
  do
    Result := (elementos @ i) @ j;
  end;
 ActualizaElemento(i,j : INTEGER; elem : T) is
 -- Al elemento de la posición (i,j) le da el valor elem
 require
     rango: 0 < i  and i <= filas and 0 < j and j <= columnas
 do
```

Cláusula de indexación. Para ser utilizada por las herramientas de recuperación de información y localización de clases.

La clase MATRIZ es una clase genérica con parámetro genérico formal T restringido a conformar con NUMERIC.

Cláusula de creación. El método creaMatriz se utiliza como método de creación de la clase.

Una de las cláusulas de recursos. Se señala separada para indicar que el atributo elementos no se exporta a nadie

Otra cláusula de recursos que agrupa a todos los que son accesibles por los clientes de la clase. Los atributos son accesibles en sólo lectura.

Signatura del método

precondiciones

definición del método (implementación)

postcondiciones

La variable predefinida **Result** contiene el valor que devolverá la función al terminar de ejecutarse

```
  (elementos @ i).put(elem,j);
end;

-- Operaciones con Matrices
suma, infix "+"(otra : like Current) : like Current is
-- Adiciona la matriz actual con la "otra" y retorna el resultado
require
  -- (n x m) + (n x m)
  dimensiones : filas = otra.filas and columnas = otra.columnas
do
   !!Result.creaMatriz(filas, columnas);
   -- ...
ensure
   dimensiones_resultado : Result.filas = filas and Result.columnas = columnas
end;
-- ... idem para la resta
mult, infix "*"(otra : like Current) : like Current is
-- Multiplica la matriz actual con la "otra" y retorna el resultado
require
  -- (n x m) * (m x k)
  dimensiones : columnas = otra.filas
do
   !!Result.creaMatriz(filas, otra.columnas);
    -- ...
ensure
   -- dimensiones de la matriz resultante (n x k)
   dimensiones_resultado : Result.filas = filas and Result.columnas =
otra.columnas
 end;
invariant
   dimensiones : filas > 0 and columnas > 0
end – MATRIZ class
```

- declaración de varios nombres para un método (suma y "+")
- declaración de un método como un operador infijo (**infix** "+")
- declaración de tipos por asociación (**like**)

instrucción de creación (!!)
se invoca el método de creación
declarado en la cláusula **creation**.

invariante de la clase

Entornos de desarrollo basados en Eiffel

Los entornos de desarrollo basados en Eiffel más conocidos son los entornos de ISE Eiffel, Visual Eiffel y Small Eiffel. ISE (*Interactive Software Engineering*)[1] ha construido un entorno de trabajo llamado Eiffel Bench tanto para Windows95/NT como para plataformas de estaciones de trabajo con interfaces gráficas basadas en Motif. ISE Eiffel es un compilador y entorno de trabajo que incluye bibliotecas de clases reutilizables, herramientas para manejar las clases, el sistema, herramientas de recuperación de la información de clases, de análisis y diseño de clases (EiffelCASE) y de ingeniería inversa. Consta de un intérprete que permite la ejecución del sistema, la depuración simbólica, etc. La arquitectura del compilador está basada en la tecnología *Melting Ice* [2 pp. 1145]. Por otra parte, permite la generación de aplicaciones ejecutables independientemente del entorno, mediante un enlace con los compiladores de C++: Visual C++ o Borland C++.

Small Eiffel es un compilador de Eiffel de libre distribución para UNIX, DOS, OS/2 y con versiones *stand-alone* para Macintosh desarrollado por un grupo de investigación de Loria[2]. Small Eiffel es un compilador que genera código C y se integra con GNU para obtener las aplicaciones ejecutables.

Por su parte, Visual Eiffel es también un compilador y entorno de trabajo desarrollado por Object-Tools[3]. Visual Eiffel está disponible para Windows95/NT y recientemente también para linux. Incluye

[1] http://www.eiffel.com
[2] http://SmallEiffel.loria.fr/
ftp://ftp.loria.fr/pub/loria/genielog/SmallEiffel

además del compilador y el ambiente de programación otras prestaciones como, por ejemplo, generación automática de documentación de clases, biblioteca de clases reutilizables, una herramienta RAD de nombre *Display Machine*, etc.

•••

BON: una metodología de Ingeniería de Software orientada al objeto

BON es una metodología que se inscribe en la categoría de metodologías puramente orientadas al objeto. Tiene su origen en la metáfora de la programación por contrato [2] junto con algunas características del modelo de objetos que propone Eiffel [1] y diversas influencias del proyecto europeo ESPRIT II (*Research and Development Program*).

BON es un acrónimo que significa *Business Object Notation*. Como metodología, aunque generalmente aparece ligada a Eiffel, su concepción es independiente del lenguaje pero indiscutiblemente muchas de sus técnicas encuentran un equivalente inmediato en Eiffel en el proceso de implementación. Esto hace que se facilite la construcción de herramientas CASE orientadas a este lenguaje en particular. ISE Eiffel, por ejemplo, soporta BON con la herramienta EiffelCASE.

BON es un método y una notación para análisis y diseño orientados al objeto de sistemas de alto nivel que hace énfasis en permitir un desarrollo sin transiciones[4], en hacer posible la reusabilidad a gran escala y en reflejar la confiabilidad necesaria para hacer que los componentes reusables se acepten y utilicen por la industria del software.

Sus principios fundamentales son:
- Simplicidad
- Desarrollo “sin costuras” (sin transición)
- Software contractual
- Reversibilidad y
- Escalabilidad

El desarrollo sin costuras es el principio de utilizar un conjunto consistente de conceptos y notaciones a través del ciclo de vida del software, evitando los problemas de impedancia de los métodos tradicionales. Una ventaja de mantener este principio viene del hecho de que los principales esfuerzos en el desarrollo del software se emplean, no en nuevos desarrollos sino, en mantener el software. Una metodología en la que no se encuentre un salto entre las diferentes etapas está preparada para reflejar con mayor facilidad y claridad el proceso de cambio. Otra ventaja muy importante de mantener este principio es que facilita los procesos de traducción automática. Las metodologías que promueven un proceso de desarrollo sin costuras se ven más como el soporte intelectual necesario a través del proceso completo de construcción del software que como la especificación de cada una de las etapas separadas del ciclo de vida del software. Unido a esto, está la presencia del principio de simplicidad. Este principio indica que hay que minimizar el número de conceptos utilizados.

El principio de reversibilidad complementa el proceso sin costuras, garantizando que los cambios realizados en cualquier paso del proceso, incluso en la implementación o el mantenimiento, puedan reflejarse hacia atrás, a los primeros pasos, incluyendo el análisis. De esta manera, se garantiza el estado consistente del proyecto. Esto significa que la continuidad debe funcionar en ambas direcciones. Si esto no fuera posible o demasiado difícil de realizar en la práctica, los primeros niveles de modelado pasan a ser obsoletos dejando solamente el código fuente de la implementación como especificación del sistema, con todos los inconvenientes que esto supone.

Desde la perspectiva del software contractual, se observa la construcción de sistemas como una sucesión de contratos precisos entre sus módulos para garantizar confiabilidad y consistencia. La relación de una clase y sus clientes pasa a verse como un acuerdo formal donde cada parte expresa sus derechos y

---

[3] http://www.object-tools.com/

[4] continuo, sin costuras, sin transición; del término inglés *seamless*.

obligaciones. Con este principio, la especificación de un gran sistema está distribuida entre todas sus partes componentes. Las responsabilidades de cada abstracción (clase) están especificadas por contratos expresados en función de otras abstracciones. Esto garantiza que el sistema y su especificación se moverán acorde cuando el sistema evoluciona, en lugar de divergir gradualmente.

El principio de escalabilidad se manifiesta en términos de la capacidad de soportar un formalismo para representar grupos de clases progresivos y soportar la división del problema basado en capas de abstracción como manejo de la complejidad estructural.

En un método sin costuras de análisis y diseño orientado al objeto, la continuidad entre los pasos conduce a la conclusión de que no existe una clara distinción entre qué es lo que realmente pertenece al análisis y qué pertenece al diseño. En realidad, la idea de BON es caracterizar los objetivos y procesos específicos del análisis y del diseño pero asegurando que el paso de uno a otro no esté regido por ninguna condición, que las notaciones y conceptos que se emplean en cada una de las etapas sean consistentes en el sentido de que el mismo concepto solamente evoluciona en la medida que se avanza, y que apoyándose en la reversibilidad, se garantice la iteratividad en el ciclo de vida del software.

El objetivo del análisis es poner un cierto orden en nuestra concepción del mundo real. El propósito es simplificar, dominar la complejidad mediante la reformulación del problema. En el análisis deben eliminarse redundancias en las especificaciones, ruidos; encontrar inconsistencias, posponer decisiones de implementación, dividir el espacio del problema, tomar un cierto punto de vista y documentarlo. El método BON considera que el análisis se convierte en diseño cuando se toman decisiones de implementación, cuando se introducen grados de protección para la información o cuando se introducen clases que no están relacionadas con el espacio de objetos del problema. El diseño es un proceso que toma por entrada una representación del problema y la transforma en una representación de la solución, regresando al análisis si es necesario. En el diseño, las clases obtenidas en el análisis se extienden, generalizan y se transforma su representación en un esquema fácilmente traducible a un lenguaje de programación orientado al objeto. Se añade la especificación de nuevas clases que aparecen como interfaz con el exterior del sistema, otras que son de utilidad básica para la construcción del sistema (p.e. clases contenedoras, etc.) y otras que son consideradas como clases de aplicación que tratan con información dependiente de la máquina o del sistema, con persistencia de objetos, manejo y recuperación de errores, etc.

En BON se modelan tanto el espacio del problema como el espacio de la solución como abstracciones de datos que encapsulan servicios. En el modelado, aunque lo primero en que se piense sea en un servicio, debe buscarse la abstracción (clase) subyacente que representa a aquellos individuos que ofrecen tal servicio. El primer principio es considerar las clases como una representación de tipos de datos abstractos que tienen un estado interno y ofrecen servicios, opuesto al criterio “las cosas que hacen” que está más cerca de las técnicas procedurales.

Las clases no se ubican aisladamente sino que se agrupan en *clusters*. Durante el análisis, los *clusters* ayudan en cuanto agrupan clases de acuerdo con un criterio de proximidad basado en funcionalidad de subsistema, en nivel de abstracción o en el punto de vista del usuario final. En el diseño, son utilizados frecuentemente como una técnica estructurada para visualizar selectivamente las conexiones entre las clases. Es muy común comenzar por un *cluster* general y luego ir determinando otros después de haber hecho ciertos agrupamientos de clases. Es importante ubicar el lugar de una clase dentro de la estructura completa. Esto implica que encontrar clases relacionadas es más significativo que encontrar una clase aislada. El uso de los *clusters* es una técnica para dotar de escalabilidad al método.

Cada etapa, análisis y diseño, en BON aporta al modelo general desde dos puntos de vista: arquitectura y comportamiento. El resultado es un modelo subdividido, dependiendo de lo que se describe, en modelo estático y modelo dinámico. Las actividades globales que describe el método para ir creando dichos modelos se describen en la tabla 1 mientras que en las tablas 2 y en la figura 1 se describen los elementos gráficos más generales que se utilizan en la descripción del modelo.

Como una tarea fundamental en el método aparece que antes de considerar completo el análisis y diseño y pasar a la fase de implementación, se debe comenzar un proceso de generalización. En este proceso debe analizarse si la arquitectura del sistema es suficientemente general para maximizar futuras

reutilizaciones. De esta manera, del conjunto de clases obtenidas se factorizan recursos comunes en clases de alto nivel (por herencia o por genericidad) y las clases existentes se convierten en versiones especializadas de estas últimas.

TAREA	DESCRIPCIÓN	ESQUEMA
1. Encontrar clases	Delinear la frontera del sistema. Encontrar subsistemas, metáforas de los usuarios, casos de uso.	Tabla de Sistema (System Chart), Tablas de Escenarios (scenario Charts)
2. Clasificar	Listar clases candidatas. Crear un glosario de términos técnicos.	Tabla de Clusters (Clusters Chart)
3. Encontrar clusters	Seleccionar clases y agruparlas en clusters. Clasificar, esbozar colaboraciones fundamentales entre clases	Tabla de Sistema (System Chart), Tabla de Clusters (Clusters Chart), Arquitectura (Static Architecture), Diccionario de clases (Class Dictionary)
4. Definir los recursos de las clases	Definir las clases. Determinar comandos (¿Qué servicios pueden solicitar otras clases a esta?), consultas (¿Qué información pueden preguntar otras clases a esta?) y restricciones (¿Qué conocimiento debe mantener la clase?)	Tablas de Clases (Class Charts)
5. Seleccionar y describir escenarios de objetos	Esbozar el comportamiento del sistema. Identificar eventos, creación de objetos y escenarios relevantes derivados de la utilización del sistema	Tablas de Eventos (Event Charts), Tablas de Escenarios (Scenario Charts), Tablas de Creación (Creatio Charts), Escenarios de Objetos (Object Scenarios)
6. Especificación de las condiciones contractuales.	Definir los recursos públicos. Especificar tipos, signaturas y contratos formales.	Interfaz de clases (Class Interfaces), Arquitectura (Static Architecture)
7. Refinar el sistema.	Encontrar nuevas clases de diseño, adicionar nuevos recursos.	Interfaz de clases (Class Interfaces), Arquitectura (Static Architecture), Diccionario de clases (Class Dictionary), Tablas de Eventos (Event Charts), Escenarios de Objetos (Object Scenarios)
8. Incrementar el potencial de reusabilidad	Generalizar. Factorizar comportamiento común.	Interfaz de clases (Class Interfaces), Arquitectura (Static Architecture), Diccionario de clases (Class Dictionary)
9. Indexar y Documentar	Documentar explícitamente la descripción de una clase mediante la cláusula de indexado. Completar la información de documentación en el diccionario de clases.	Interfaz de clases (Class Interfaces) Diccionario de clases (Class Dictionary)
10. Evolucionar la arquitectura del sistema	Completar y revisar el sistema. Producir una arquitectura final con un comportamiento del sistema.	Modelos estático y dinámico finales de BON. Todos los esquemas completados.

Tabla 1

Tabla de Sistema (System Chart): Definición del sistema y lista de los clusters asociados. Solamente una Tabla de sistema por proyecto. Los subsistemas se describen mediante su correspondiente Tabla de Cluster.

Tablas de Clusters (Cluster Charts): Definición de clusters y lista de las clases asociadas y subclusters, si los hay. Un cluster representa un subsistema completo ó sólo un grupo de clases.

Tablas de Clases (Class Charts): Definición de las clases de análisis en términos de comandos, consultas y restricciones, de forma que sea entendible para los expertos en el dominio y personal no técnico.

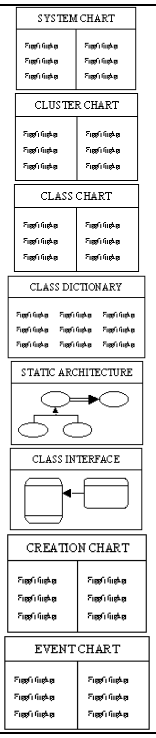
Diccionario de Clases (Class Dictionary): Una lista alfabéticamente ordenada de todas las clases del sistema, mostrando el cluster al que pertenece cada clase y una breve descripción. Debe poder ser generada automáticamente de las Tablas de Clase y de Interfaz.

Arquitectura (Static Architecture): Conjunto de diagramas que posiblemente representan clusters anidados, encabezamientos de clases y sus relaciones. Una vista del sistema (con posibilidades de hacer zoom)

Interfaz de Clases (Class Interfaces): Definiciones tipadas de las clases con la signatura de los recursos y contratos formales con un lenguaje basado en el cálculo de predicados. Vista detallada del sistema

Tablas de Creación (Creation Charts): Lista de las clases que están a cargo de crear instancias de otras clases. Normalmente se hace una para el sistema pero si se desea se puede incluir una por subsistema.

Tablas de Eventos (Event Charts): Conjunto de eventos externos (estímulos) que disparan algún comportamiento interesante del sistema y el conjunto de respuestas del sistema. Puede ser repetido para cada subsistema.



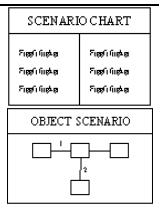
Tablas de Escenarios (Scenario Charts): Lista de los escenarios de objetos utilizados para mostrar algún comportamiento interesante y representativo del sistema. Los subsistemas pueden contener Tablas de Escenario locales. **Escenarios de Objetos** (Object Scenarios): Diagramas dinámicos que muestran comunicaciones relevantes entre objetos para algunos o todos los escenarios que se describen en las Tablas de Escenarios.	

Tabla 2

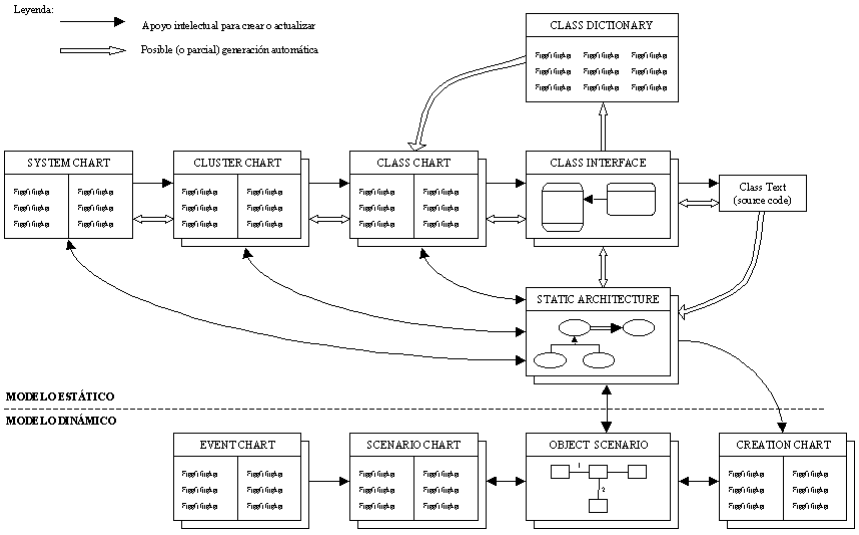


Figura 1

…

Referencias

(1) Meyer, B. Eiffel: the language. Prentice Hall, 1992
(2) Meyer, B. Object Oriented Software Construction. Second edition, Prentice Hall International 1997.
(3) Meyer, B. Tools for a new culture: Lessons from the design of the Eiffel Libraries. Communications of the ACM vol. 33, nº 9, Sept. 1990.
(4) Nerson, J-M. Applying Object-Oriented Analysis and Design. Communications of the ACM vol. 35, nº 9, Sept. 1992.
(5) Waldén, K. & Nerson, J-M. Seamless Object-Oriented Software Architecture, analysis and design of reliable systems. Object-Oriented Series, Prentice Hall International, 1995.