Una herramienta para la enseñanza de patrones en Ingeniería del Software

Macario Polo, Juan Ángel Gómez, Mario Piattini y Francisco Ruiz
Escuela Superior de Informática – Universidad de Castilla-La Mancha
Paseo de la Universidad, 4
13071-Ciudad Real
macario.polo@uclm.es

Resumen.
Se presenta una herramienta que genera una aplicación totalmente ejecutable a partir de un diagrama de clases dibujado con Rational Rose. La herramienta considera el diagrama como la estructura de la capa de Dominio de un sistema de tres capas. A partir de esta idea, la herramienta genera las capas adyacentes (Presentación y Almacenamiento), dándole ciertas funcionalidades suministradas por un conjunto de patrones de diseño.

1. Introducción.

Los patrones de diseño son parte de los contenidos de la asignatura "Ingeniería del Software II", del 5º curso de la Ingeniería en Informática de la Universidad de Castilla-La Mancha. La asignatura consta de 9 créditos, 6 de los cuales corresponden a teoría y 3 a clases de laboratorio. Entre las herramientas que los alumnos utilizan en los laboratorios se encuentra Rational Rose, que es utilizada como base para la construcción de una aplicación ejecutable que, independientemente de que funcione correctamente o no, debe poseer un buen diseño, arquitectura robusta, etc., aspectos éstos que son los más valorados a la hora de determinar la calificación. La idea de este trabajo práctico es que los alumnos apliquen a un caso concreto los contenidos teóricos de la asignatura, de los que los patrones de diseño son una buena parte.

Durante las clases teóricas, se pone especial énfasis en la necesidad de dotar a las aplicaciones de una arquitectura robusta, y suelen utilizarse como ejemplos aplicaciones con tres capas (normalmente Presentación, Dominio y Almacenamiento). Una aplicación de tres capas bien elegida posee la versatilidad suficiente como para incluirle los elementos necesarios para presentar ejemplos de prácticamente todos los patrones de diseño que se presentan durante la asignatura. Así, por ejemplo:

1. La propia utilización de una arquitectura multicapa es ya un buen patrón, que dota a las aplicaciones de gran escalabilidad, que permite la reutilización, etc. Además, las clases y paquetes de las aplicaciones de tres capas bien construidas poseen valores adecuados de cohesión y acoplamiento, cuyos equilibrados valores son probablemente dos de los más básicos principios de un diseñador de software.
2. El hecho de que los objetos de la capa de Presentación necesiten "refrescarse" para mostrar los cambios de estado experimentados por los objetos de la capa de Dominio, permite la incorporación a la aplicación de clases que constituyen implementaciones del patrón Observer.
3. La utilización de una base de datos que almacena (en forma de registros) las instancias persistentes procedentes de clases de la capa de Dominio, permite:
   - 3.1 La utilización de alguno de los muchos patrones existentes para hacer corresponder clases y tablas.
   - 3.2 La utilización de patrones para decidir a qué clases deben asignarse las responsabilidades relacionadas con la persistencia de los objetos.
   - 3.3 La utilización de patrones para transformar el diagrama de clases de la capa de Dominio (o parte de él) al esquema físico de la base de datos.

3.4 La utilización de algún patrón que centralice el acceso de los objetos de la capa de Dominio a la base de datos (un Broker o Agente).

Desde luego, resulta imposible construir en el laboratorio todas las posibles variantes de la aplicación utilizando todos los posibles patrones de diseño, incluso aunque ésta sea sencilla. También es difícil conseguir que los estudiantes implementen, aunque sea fuera del laboratorio, todas las posibles variantes. Pero el caso es que, observando las reglas de transformación y diseño que, más o menos, rigen el mecanismo de utilización de los patrones comentados en los puntos anteriores, se observa que el proceso de aplicación de dichos patrones es –para una persona- un proceso muy automático que puede, además, ser automatizado por una herramienta.

En la Escuela Superior de Informática de Ciudad Real hemos construido una herramienta que, a partir de un diagrama de clases dibujado con Rational Rose, permite al usuario generar código directamente ejecutable. El código se genera de acuerdo a un conjunto de patrones, que el alumno ha podido elegir de un conjunto de patrones disponibles. La generación de código tarda sólo unos pocos segundos, lo que permite al alumno generar en muy poco tiempo lo que habitualmente tardaría horas o días, y sin ningún error. Esto permite aprovechar más aún las horas de laboratorio, ya que se libera tiempo que puede dedicarse a la aplicación práctica de otros contenidos presentados en las horas de teoría.

En este artículo presentamos algunos detalles del diseño e implementación de esta herramienta, así como su forma de uso y sus posibilidades.

2. Aspecto de la herramienta.

La Figura 1 muestra el aspecto de la ventana principal de nuestra herramienta. En ella, el alumno selecciona un fichero que contenga un modelo de Rational Rose. El conjunto de clases contenido en dicho modelo se muestra en la lista de la izquierda, debiendo el usuario seleccionar las clases para las que se desea generar código.

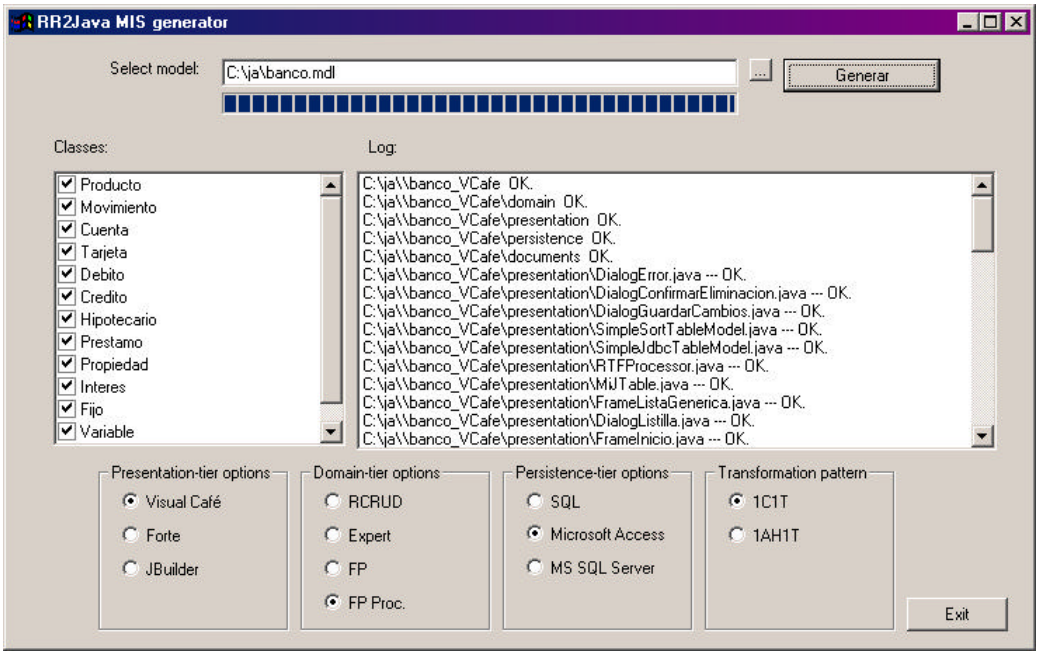

Figura 1. Pantalla principal de la herramienta.

El alumno debe saber que la herramienta ubicará las clases seleccionadas en la capa de Dominio de la aplicación generada, por lo que no es conveniente seleccionar clases que representen ventanas, etc. Para cada una de las tres capas para las que se va a generar código pueden elegirse diversas opciones (situadas en la parte inferior de la ventana):

1. Para la capa de Presentación puede elegirse el entorno de desarrollo para el que se generará código.
   - 1.1 Aunque se observan varios entornos de lenguaje Java, ocurre que los Frames construidos con, por ejemplo, Visual Café no son editables con Forte o JBuilder, razón por la que los posibles entornos se colocan como opciones excluyentes.
   - 1.2 Se genera una pantalla "tipo ficha" por cada clase seleccionada. Además, se hace que cada clase de la capa de Presentación conozca a su instancia correspondiente de la capa de Dominio (por ejemplo: una pantalla que muestra los datos de un empleado conoce a su objeto Empleado correspondiente).
   - 1.3 Se generan un conjunto de ventanas adicionales y "constantes", cuya implementación no depende o depende muy poco de la aplicación que se está generando. Ejemplos de estas ventanas son un diálogo para mostrar mensajes de error, una ventana que muestra listas de registros, una ventana que pide confirmación antes de borrar o modificar, etc.
   - 1.4 Todas las pantallas generadas poseen un conjunto de operaciones que permiten invocar operaciones de persistencia sobre los objetos de la capa de Dominio (las pantallas "tipo ficha", por ejemplo, tienen un botón "Guardar", que guarda el objeto actualmente mostrado en la base de datos, mediante una llamada al método *insert* o *update* –depende del caso- del correspondiente objeto de la capa de Dominio).
2. La capa de Dominio se genera mediante una "traducción directa" de las clases seleccionadas al lenguaje de programación considerado. Además, para cada una de estas clases se genera el siguiente conjunto de métodos:
   - 2.1 Un constructor sin parámetros, que da a los campos de la clase valores por defecto (por ejemplo, el valor cero a los *long*).
   - 2.2 Un constructor con parámetros que permite materializar objetos (esto es, construir instancias de clases a partir de los registros almacenados en la base de datos). Los valores de los parámetros pasados a este constructor se corresponden con los valores de las columnas que forman la clave principal del registro que queremos materializar.
   - 2.3 Métodos *insert*, *delete*, *update* y *delete*, que permitirán actualizar el objeto en la base de datos. Esto constituye una implementación del patrón CRUD (operaciones para hacer Create, Read, Update y Delete), expuesto por Yoder [4].
3. Para la capa de Persistencia se genera siempre un Agente [1] cuya implementación es siempre constante, y que ejecuta instrucciones SQL sobre la base de datos. Existen cuatro formas de asignar y generar las responsabilidades de persistencia:
   - 3.1 Mediante el patrón Experto, cada clase persistente define e implementa sus operaciones de persistencia. Es decir, en ellas reside completamente el código de las operaciones *insert, delete,* etc. Esta alternativa tiene la desventaja de que hace a las clases de la capa de Dominio completamente dependientes del gestor de base de datos utilizado.
   - 3.2 Para desacoplar a las clases de la capa de Dominio del sistema gestor de base de datos, las operaciones de persistencia pueden delegarse a "fabricaciones puras" [2]. De este modo, si se cambia de gestor de base de datos, sólo se necesitará modificar las clases asociadas (las fabricaciones puras, que residen en la capa de Almacenamiento), y no las de Dominio, que son mucho más complejas puesto que son las que verdaderamente implementan la solución del problema propuesto (que habitualmente será mucho más que gestionar persistencia). El sencillo ejemplo con que ilustramos en clase esta dependencia del gestor de base de datos es que, si se usa Access, deben usarse las palabras *true* o *false* para guardar valores en columnas boolea-

nas, mientras que se utilizan 1 y 0 con SQL Server. Existen, de todos modos, varias formas de implementar las operaciones de persistencia en las fabricaciones puras, como hacerlas estáticas o no (Figura 2). Nuestra herramienta siempre genera como estáticas estas operaciones. En cualquier caso, y aún habiendo decidido que las operaciones de persistencia en las fabricaciones puras sean estáticas, nuestra herramienta ofrece dos posibilidades: que la fabricación pura genere y directamente ejecute la instrucción SQL (opción "FP" en la ventana de la Figura 1), o mediante generación de sentencias preparadas (opción "FPProc" de la Figura 1).

3.3 Las operaciones de persistencia pueden también ser asignadas mediante el patrón RCRUD [3]. RCRUD (Reflective Create, Read, Update & Delete) es una clase, totalmente reutilizable, que genera mediante introspección (*Reflection)* las operaciones de persistencia de cualquier clase. El único requisito para utilizar RCRUD es que las clases persistentes sean especializaciones de RCRUD.

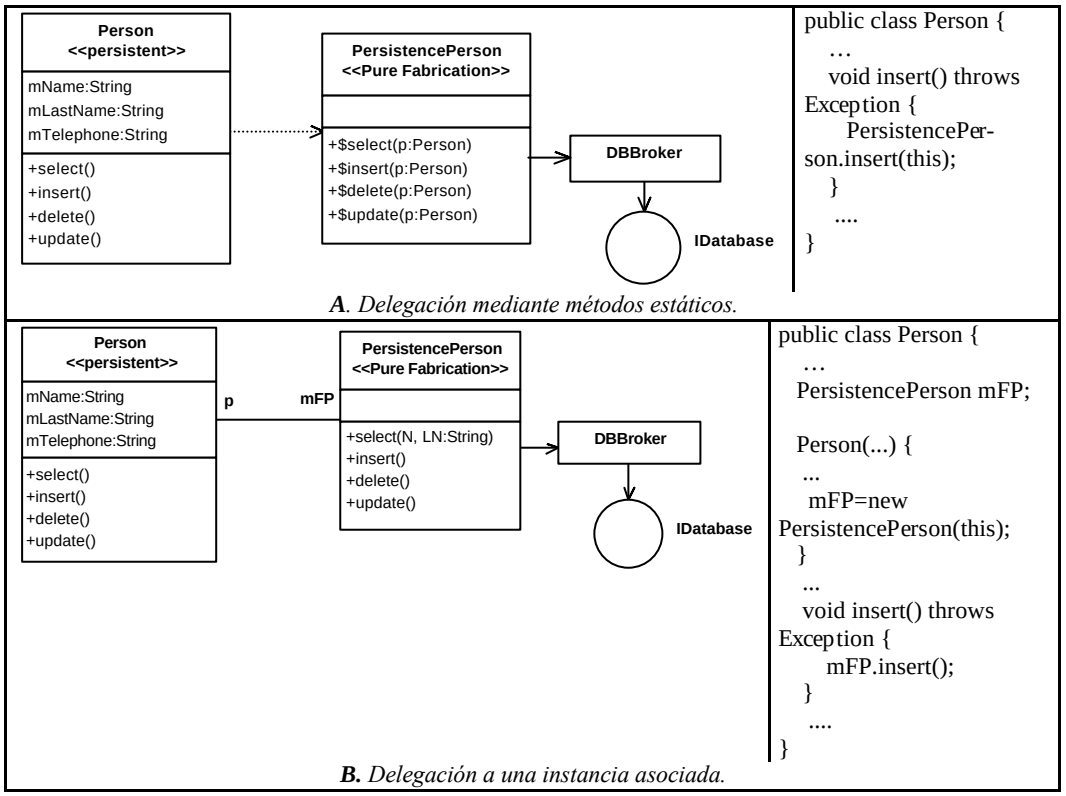

Figura 2. Posibles formas de delegar las operaciones de persistencia.

4. Se genera, además, la base de datos resultante de transformar el diagrama de clases mediante uno de los siguientes patrones:
   - 4.1 Una clase, una tabla
   - 4.2 Un árbol de herencia, una tabla
   - 4.3 Tenemos pendiente de implementación la generación de la base de datos mediante el patrón "Un camino de herencia, una tabla.

     La siguiente figura muestra tres esquemas relacionales (B, C y D) obtenidos de la aplicación de estos tres

patrones de transformación al diagrama de clases (A).

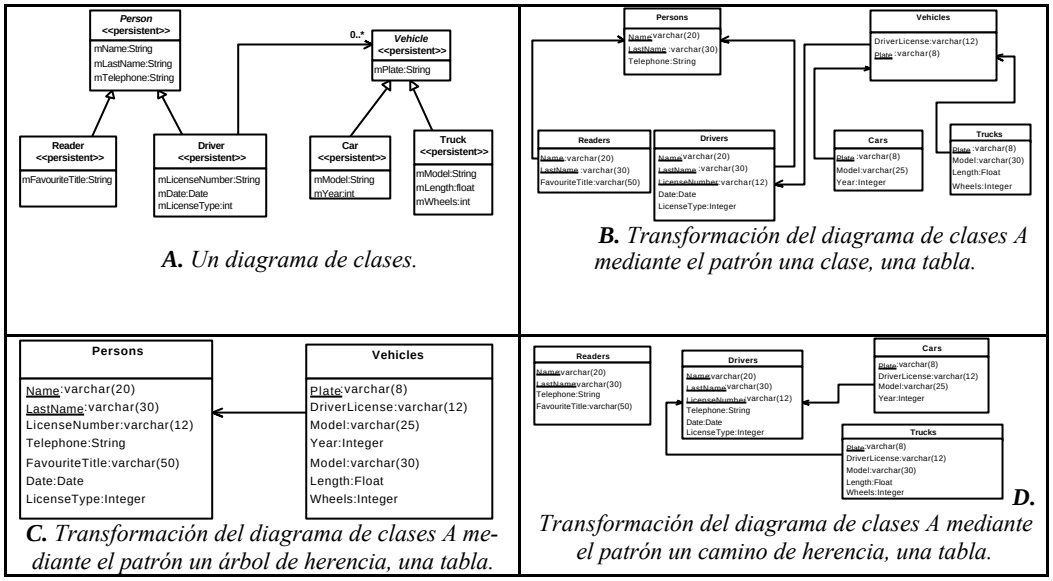

Figura 3. Tres posibles transformaciones a relacional de un mismo modelo de clases.

La base de datos puede generarse directamente en un fichero Access, en una base de datos de un servidor SQL Server, o bien como un programa de definición de datos en SQL 92.

Con estas consideraciones, los objetos que intervienen y se generan para gestionar en cierto escenario los datos de una instancia de clase Persona son los mostrados en la Figura 4 (suponiendo que se han seleccionado fabricaciones puras para delegar las operaciones de persistencia).

3. Utilización de la herramienta en los laboratorios.

La herramienta es capaz de generar en segundos varias versiones diferentes de una aplicación que gestiona una base de datos. Además de ser ejecutable, la aplicación generada puede ser importada desde Rational Rose o alguna otra herramienta (como Poseidon, de Gentleware), de manera que puede cargarse su diagrama de clases para realizar comparaciones de los diferentes diseños, etc.

La Figura 5 muestra un fragmento de la estructura de la aplicación obtenida al generar código para las clases que hemos recuadrado, y que fueron originalmente dibujadas con Rational Rose:

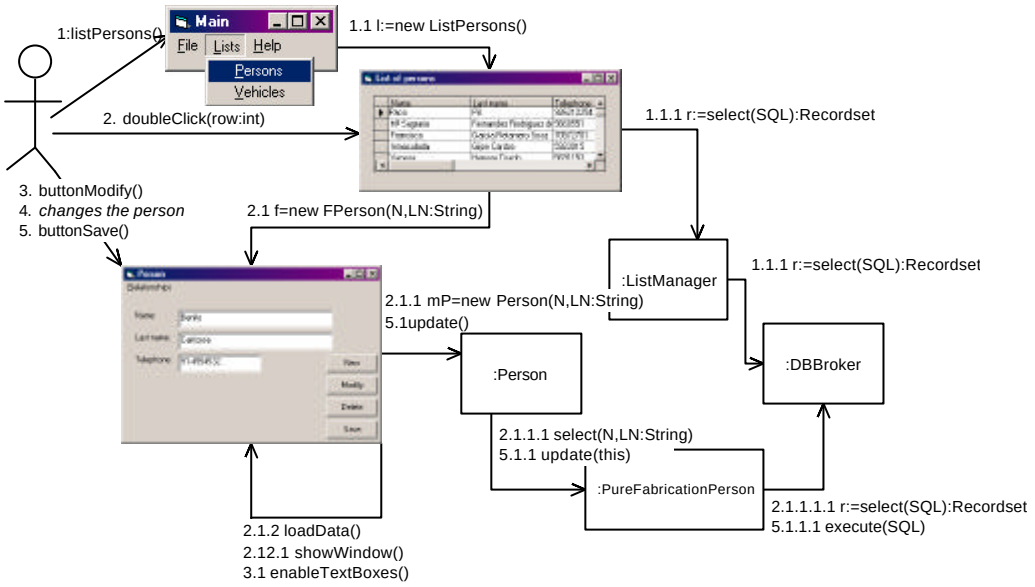

Figura 4. Objetos que intervienen en una aplicación generada por la herramienta, en este ejemplo para gestionar cierto escenario de una persona.

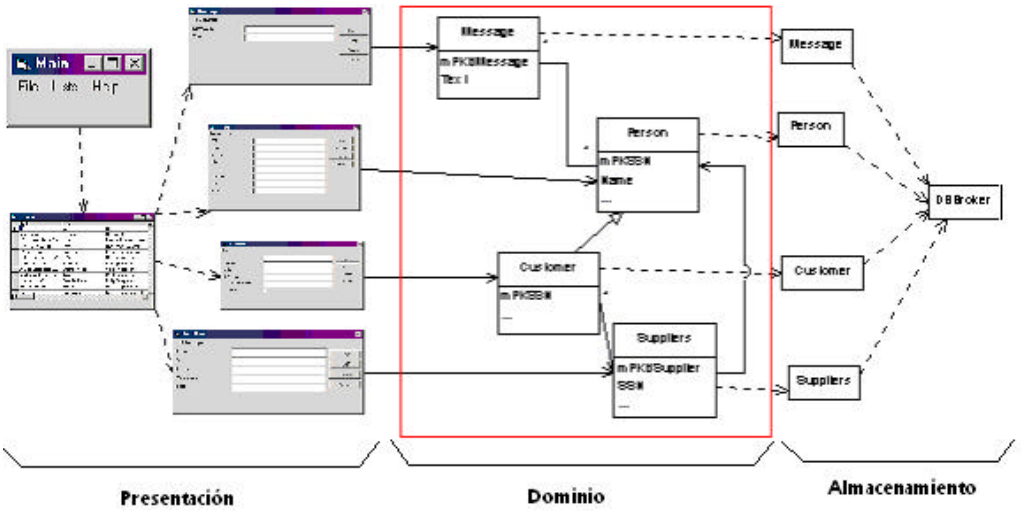

Figura 5. Fragmento de la estructura de una de las aplicaciones generadas.

Cuando el usuario ejecuta la aplicación generada, se encuentra con una pantalla principal que le da a acceso a los listados de todas las tablas existentes en la base de datos. Haciendo doble clic en una de las filas de la lista, se abre la correspondiente pantalla de tipo ficha, que permite la ges-

tión del registro asociado a través de la capa de Dominio.

4. Arquitectura de la herramienta generadora.

La Figura 6 muestra un fragmento del diseño de la herramienta generadora: incorpora la definición de un metamodelo que permite representar aplicaciones de tres capas. La clase "Three-tier application" contiene tres referencias a objetos que se encargan de la generación del código para la capa de Presentación, Dominio y Almacenamiento mediante la implementación de los tres interfaces representados en la figura. De este modo, la adición a la herramienta de un generador de código para otro lenguaje, base de datos u otro tipo de entorno precisa únicamente la construcción de una clase que implemente el interfaz deseado. En estos momentos estamos implementando dos nuevos generadores que generarán Servlets y JSPs en la capa de Presentación.

5. Conclusiones y trabajos futuros.

En este artículo se ha presentado una herramienta que resulta de gran ayuda para la enseñanza de patrones en asignaturas de Ingeniería del Software. La herramienta toma un diagrama de clases dibujado con Rational Rose y genera código multicapa de alta calidad. La gran ventaja aportada por la herramienta es que genera en muy poco tiempo diferentes aplicaciones, cada una con un diseño diferente (dependiendo de los patrones elegidos), lo que libera a los estudiantes de tediosas tareas de implementación de ejemplos, dejando más tiempo para la aplicación práctica de otros contenidos teóricos de la asignatura.

Referencias.

[1] Buschman F., Meunier R., Rohnert H., Sommerlad P. and Stal M. (1996). A System of Patterns: Pattern-Oriented Software Architecture. Addison Wesley.

[2] Larman C. (1998). Applying UML and Patterns. Upper Saddle River, NJ: Prentice-Hall.

[3] Polo M, Piattini M and Ruiz F. (2001). RCRUD: Reflective Create, Read, Update and Delete. Proc. of the Sixth European Conference on Pattern Languages of Programs (EuroPlop'2001). Irsee, Alemania. También disponible en (9 de mayo de 2002): http://www.inf-cr.uclm.es/www/mpolo

[4] Yoder, JW. (2001). *Patterns for making business objects persistent in a relational database.* Tutorial at the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Tampa Bay, Florida, USA.

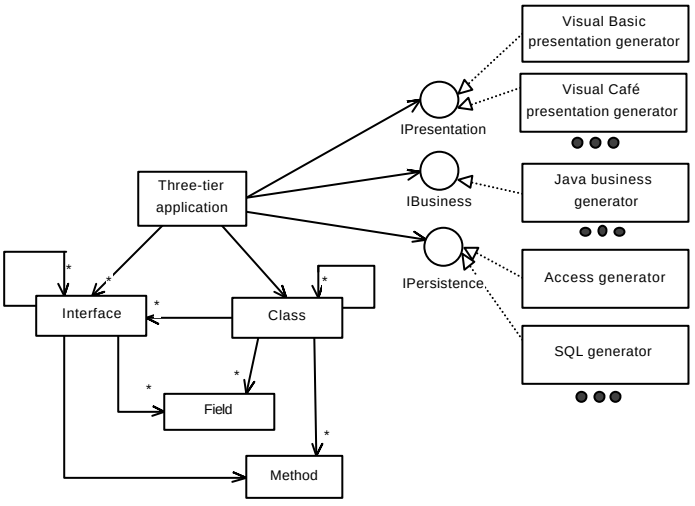

Figura 6. Arquitectura de la herramienta generadora.