Universidad Autónoma de Madrid

Trabajo de Fin de Grado

Un lenguaje de modelado para la creación de anotaciones Java

Autora:
Irene Córdoba Sánchez

Tutor:
Juan de Lara Jaramillo

Mayo 2014

Resumen

Desde su introducción en Java 5, las anotaciones han pasado de ser una herramienta útil a jugar un papel central en muchos proyectos software populares, como por ejemplo la *Java Persistence API* (JPA), API para la persistencia de objetos Java, o el *framework* de construcción de aplicaciones Spring.

Las anotaciones raramente se conciben de forma aislada, sino que forman un conjunto con interdependencias y restricciones de integridad entre ellas y los elementos que anotan; sin embargo, Java no ofrece un mecanismo para la expresión de dichas restricciones. Además, el soporte sintáctico para la definición de anotaciones es en sí mismo muy limitado, dado que se reutilizan construcciones Java para otros propósitos.

Para suplir estas carencias se ha construido el *lenguaje de modelado de dominio específico* **Ann**, que permite hacer explícito el modelo conceptual de restricciones que hay detrás de un conjunto de anotaciones. Se proporciona así mismo un entorno de modelado para definir los modelos de anotaciones con una sintaxis más acorde al resto de elementos de Java; y un generador de código Java para integrar los conjuntos de anotaciones diseñados en proyectos ya existentes en dicho lenguaje. El desarrollo de **Ann** se ha realizado mediante la *Eclipse Modeling Framework*.

En la validación de **Ann** se ha utilizado un subconjunto de anotaciones de JPA y se han expresado distintas restricciones de integridad entre ellas y con los elementos que anotaban. El resultado ha sido muy satisfactorio, al haber conseguido expresar el modelo conceptual subyacente casi al completo.

Respecto a otras herramientas similares, **Ann** ha demostrado tener un valor añadido en distintos ámbitos de aplicación comunes. Además se dispone de un amplio abanico de posibles líneas de trabajo futuro, como por ejemplo aplicaciones en la ingeniería inversa o extensiones al lenguaje, al ser las anotaciones una característica relativamente reciente y el soporte actual de Java muy limitado.

Para la realización de este proyecto me fue concedida una Beca de Colaboración por el Ministerio de Educación de España.

Palabras clave

Java, anotaciones, modelado, *Lenguajes de Dominio Específico*, generación de código.

Abstract

Since their addition in Java 5, annotations have grown from a useful tool to play a central role in many popular software projects; like the Java Persistence API (JPA) or the Spring framework.

Annotations are rarely conceived in an isolated way; instead, they are usually part of a set, with dependencies and integrity constraints within them and the elements they are attached to. However, there is no mechanism in Java to express those constraints. Besides, the syntactic support for the definition of annotations is itself very limited, given that some Java constructions are reused for other purposes.

To overcome these defficiencies, I have built the *domain-specific modeling language* **Ann**, which allows making explicit the conceptual model of the constraints behind a set of annotations. A modeling environment is also provided to define the annotation models with a more homogeneous sintax within the rest of Java elements. In addition, a Java code generator has been implemented in order to integrate the designed sets of annotations with existing projects in such language. **Ann** has been developed using the *Eclipse Modeling Framework*.

A subset of JPA annotations has been chosen for the validation of **Ann**, and several constraints within them and with the elements they decorate have been expressed. The result has been very satisfying, given that the underlying conceptual model has been expressed almost entirely.

With respect to other similar tools, **Ann** has proven to have an added value in different shared areas of application. Regarding to possible future lines of work, there is a huge number of possibilities, such as applications in reverse engineering or language extensions, given that annotations are a feature relatively recent and the Java support is very limited nowadays.

I was granted a Collaboration Scholarship by the Spanish Ministry of Education for the development of this project.

Keywords

Java, annotations, modeling, *Domain-Specific Languages*, code generation.

Índice general

Índice de figuras

1. Introducción

Las anotaciones Java son un tipo de elemento que se introdujo en el lenguaje Java en el año 2004 [18], como parte de la versión 5.0. Según la documentación publicada en aquel entonces, surgieron en respuesta a la gran cantidad de código repetitivo que requieren muchas *Application Programming Interface*s (APIs).

Un ejemplo son un tipo de *web service* Java, los *Java API for XML Web Services* (JAX-WS), en el que es necesario un par asociado interfaz e implementación. Éstas podrían ser fácilmente generadas de forma automática por alguna herramienta si el programa estuviese «decorado» con anotaciones que indicasen qué métodos son accesibles remotamente.

También nos encontrábamos con APIs que requerían que se mantuviesen ficheros laterales de forma paralela al programa, como por ejemplo los *JavaBeans*, en los que la clase *BeanInfo* se tiene que mantener en paralelo a la *bean*; y los *Enterprise JavaBean*s (EJBs), en los que se requiere un descriptor de despliegue. Se hace evidente que hubiera sido mucho más conveniente, y también menos propenso a fallos, que la información en estos ficheros laterales se mantuviese como anotaciones en el propio programa sobre los objetos con los que están relacionados.

Algunos autores sin embargo atribuyen la aparición de las anotaciones en el lenguaje Java a la cada vez más creciente tendencia a incluir los meta-datos asociados a un programa en el propio programa en vez de mantenerlos en ficheros separados (como sucedía con los *JavaBeans*); así como la presión ejercida desde otros lenguajes de programación que ya incorporaban características similares, como C# [4].

En cualquier caso, desde su introducción las anotaciones en Java han constituido un gran éxito, como queda patente por su amplio uso en distintos proyectos de gran importancia en el mundo actual del desarrollo software. En el lado del código abierto encontramos por ejemplo los *framework*s *Seam* [26] y *Spring* [27], así como los *Object Relational Mapping* (ORM) de *Hibernate* [28]. Así mismo, otros estándares de *Sun Java* además de EJB y JAX-WS también usan anotaciones de forma clave, como por ejemplo *Java Persistence API* (JPA) [16].

Sin embargo, a pesar del gran cambio que han supuesto las anotaciones para el desarrollo software, el soporte que ofrece Java es muy limitado en cuanto a sintaxis y semántica. En el presente documento se expondrá y analizará una solución inicial a esta carencia.

Para la realización de este trabajo me fue concedida una Beca de Colaboración por el Ministerio de Educación.

1.1. Motivación

Aunque en el Capítulo 2 se realizará un estudio detallado de las anotaciones Java, veamos un poco acerca su situación actual para comprender las carencias del lenguaje, y con ello la motivación del presente texto.

La sintaxis de Java asociada a las anotaciones es definitivamente inusual. En la Figura 1.1 vemos un ejemplo de una declaración sencilla de una anotación.

```
public @interface Review {

    public static enum Grade {
        EXCELLENT,
        SATISFACTORY,
        UNSATISFACTORY
    };

    Grade grade();
    String reviewer();
    String comment() default "";
}
```

Figura 1.1. Declaración de una anotación en Java.

Para empezar, una anotación es una «interfaz especial». Es por ello que la declaración es muy similar a la de una interfaz común, con la salvedad del símbolo @ antes de `interface`. Otro hecho que llama la atención son los métodos sin argumentos que aparecen: esos son los «campos» de la anotación. Su nombre y valor de retorno indican el *nombre* y el *tipo* del miembro en cuestión. Son métodos que no pueden lanzar excepciones, pero más peculiar es aún la manera de asignar el valor por defecto. En vez de utilizar un «=» como sucede con cualquier otro campo, se utiliza la palabra clave `default` y a continuación el valor que se quiere asignar.

Este pequeño ejemplo nos muestra que la sintaxis para la definición de anotaciones en Java deja bastante que desear en cuanto a su integración y coherencia con el resto de componentes del lenguaje:

- Se utiliza para definir un tipo de atributo, los campos, una sintaxis correspondiente a otro distinto, los métodos.
- La asignación del valor por defecto es distinta a la dada en cualquier otro contexto (inicialización de variables locales, variables de instancia,...).
- Se requiere un carácter especial adjuntado a `interface` para definir el tipo de elemento; en vez de, o bien gozar de un nombre propio (por ejemplo `annotation`), o bien ser tratado como cualquier otra interfaz (que no es posible dadas sus características especiales).

Por otro lado, Java ofrece algunas opciones para añadir restricciones a una anotación. Por ejemplo, en el caso de que queramos que una anotación que hemos definido solamente se pueda utilizar en métodos, podemos expresarlo con la sintaxis de Java.

Sin embargo, este soporte pronto se muestra insuficiente. Sin ir más lejos, si miramos el propio estándar de *Sun Java* JPA encontramos la anotación `javax.persistence.Entity`, cuya documentación [15] establece, entre otros, los siguientes requisitos para una clase que esté anotada con ella:

- Tener un constructor `public` o `protected` sin argumentos.
- No ser final.
- No tener ningún método final.
- Sus campos persistentes deben ser declarados `private`, `protected` o `package-private`.

Ninguno de estos requisitos se puede expresar a día de hoy con el soporte que ofrece Java; en su lugar, o bien obtendremos errores en tiempo de ejecución, o bien tendremos que escribir extensiones al compilador que los comprueben (conocidas como procesadores de anotaciones).

Por último, las anotaciones raramente se conciben de manera aislada, sino que forman un conjunto coherente, con interdependencias y restricciones de integridad entre ellas y los elementos que anotan. Sin ir más lejos, de nuevo en el contexto JPA, la anotación `javax.persistence.Id` indica la clave primaria de una entidad, luego únicamente puede ir anotando campos que se encuentren dentro de una clase anotada con `javax.persistence.Entity`.

Con todos estos ejemplos podemos extraer dos conclusiones:

1. La sintaxis de Java para la creación de anotaciones rompe con el esquema de sintaxis general para la creación de elementos en Java, haciéndola por tanto poco intuitiva para el programador no habituado a usarlas.
2. Frecuentemente se encuentran restricciones sobre conjuntos de anotaciones que no se pueden expresar con el soporte sintáctico actual de Java, quedando relegadas a comentarios en el código o a procesadores de anotaciones.

1.2. Objetivos

Para cubrir las carencias que ya hemos comprobado que presenta el lenguaje Java actualmente en el diseño de anotaciones, se propone el *Lenguaje de Modelado de Dominio Específico - Domain-Specific Modeling Language* (DSL) [6] **Ann**. Dicho lenguaje y sus prestaciones son el objeto del presente documento, y serán analizados en profundidad en las secciones siguientes. Más adelante veremos en detalle en qué consisten este tipo de lenguajes; por ahora nos interesa los objetivos que debe cumplir en concreto **Ann**, dada la situación vista en el apartado anterior.

Como objetivo general, **Ann** pretende mejorar el soporte sintáctico de Java, permitiendo hacer explícitas restricciones y decisiones de diseño a la hora de desarrollar conjuntos de anotaciones.

Es interesante desglosar el objetivo general en puntos específicos para así poder después comprobar más concretamente cómo se van cumpliendo. El desglose de metas del DSL es por tanto:

1. Proporcionar una sintaxis acorde al diseño general de elementos Java para las anotaciones.
2. Permitir expresar restricciones acerca de conjuntos de anotaciones.
3. Proporcionar un generador de código que traslade el diseño y restricciones expresados mediante la sintaxis a código Java para su posterior integración en proyectos en dicho lenguaje.
4. Proporcionar una solución flexible y extensible que pueda evolucionar añadiendo nuevas posibles restricciones y funcionalidades.

Teniendo en cuenta el último objetivo, es importante hacer notar que **Ann** se ha restringido a las situaciones más comunes presentes en el diseño de anotaciones, así como situaciones reales en *framework*s propios de Java como los mencionados a lo largo de este capítulo. Esta restricción es lo suficientemente amplia para que constituya una herramienta útil y aplicable en diversos contextos de desarrollo, como veremos más adelante al aplicarlo a situaciones reales.

1.3. Estructura del documento

En este apartado daremos una visión global de cómo está estructurada la exposición acerca de **Ann** a lo largo del documento.

Comenzaremos introduciendo en el Capítulo 2 conceptos necesarios para entender el resto del documento. Los veremos con cierto nivel de profundidad dado que juegan un papel importante al haber sido utilizados o formar parte del contexto de desarrollo de **Ann**.

En el Capítulo 3 se estudiarán otros trabajos y propuestas relacionados con el diseño de conjuntos de anotaciones.

En el Capítulo 4 se explorará el diseño de los distintos componentes de **Ann**.

Así mismo veremos cómo se ha llevado a la práctica ese diseño en el Capítulo 5, es decir, trataremos su implementación.

Tras conocer el esqueleto e implementación de la solución propuesta, en el Capítulo 6 veremos casos reales de aplicación de la misma, analizando el valor añadido y posibles limitaciones de **Ann** en cada situación.

Por último, en el Capítulo 7 desglosaremos las distintas conclusiones que del proyecto realizado se puedan extraer; comparándolo con otras herramientas ya existentes y viendo las posibles líneas de trabajo futuro.

En otro orden pero igualmente importantes se encuentran los índices de contenido y figuras, al inicio del documento, así como los anexos técnicos para ampliar y explicar en detalle algunas secciones en concreto.

Finalmente, cerrando el documento encontramos el glosario de acrónimos y términos empleados y la bibliografía, que comprende las referencias usadas a lo largo de todo el documento.

2. Conceptos previos

En este capítulo se estudiarán las distintas tecnologías, conceptos y herramientas que han jugado un papel importante en el desarrollo de **Ann** y que serán mencionados frecuentemente a lo largo del documento.

En concreto se profundizará en los siguientes puntos:

- **Anotaciones en Java**: diseño y características principales.
- **Desarrollo Dirigido por Modelos** - *Model-Driven Development* (MDD), con especial énfasis en los lenguajes de modelado.
- ***Eclipse Modeling* Framework (EMF)**: Ecore, Xtext y Xtend.

El lector familiarizado con estos conceptos puede omitir este capítulo y pasar directamente al siguiente.

2.1. Anotaciones en Java

En el Capítulo 1 ya vimos con un pequeño ejemplo una parte de la sintaxis de declaración de anotaciones, y comentamos que las anotaciones en Java están implementadas internamente como un tipo especial de interfaz. Aunque existen muchas particularidades de las anotaciones respecto de las interfaces (que se asumen conocidas por el lector), en esta sección únicamente veremos las importantes a la hora de comprender el diseño de **Ann**.

La información contenida en este apartado ha sido extraída principalmente de la documentación oficial de Java [19] [20] y del libro de David Flanagan [1], a no ser que se indique lo contrario. Dado que recientemente ha salido la versión 8 de Java y la documentación utilizada se corresponde con la versión 7, en los apartados donde sea necesario se indicarán los cambios que esta nueva versión ha supuesto para las anotaciones.

2.1.1. Miembros

Como vimos en el Capítulo 1, los campos o miembros de una anotación se declaran de una manera especial: son métodos sin argumentos que no pueden lanzar excepciones y cuyo tipo de retorno indica el tipo del campo. Para asignarles un valor por defecto, se utiliza la palabra clave `default` y a continuación el valor a asignar (ver Figura 1.1).

Una anotación puede no tener ningún miembro (anotación de marcado), o puede tener varios. Los miembros de una anotación pueden tener los siguientes tipos (i.e. valores de retorno):

- Un tipo de dato primitivo: `short`, `int`, `long`, `float`, `double`, `boolean`, `char` o `byte`.
- `String`, `Class` o una invocación parametrizada de `Class`.
- Una anotación.
- Un *array* de uno de los tipos anteriores. No se permiten *arrays* anidados o multidimensionales.

Como hemos visto en el ejemplo de la Figura 1.1, una anotación puede llevar dentro declaraciones anidadas, como un enumerado que posteriormente se vaya a utilizar. Esta característica no se ha tenido en cuenta en el diseño de **Ann** por simplicidad, al igual que las parametrizaciones de clase.

2.1.2. *Targets*

Claramente el objetivo de una anotación es «marcar» un elemento de un programa Java con cierta cantidad de metadatos. Por tanto es necesario conocer los distintos elementos que pueden anotar. En adelante a dichos elementos los llamaremos *target*s de una anotación.

Una anotación se puede emplear en la declaración de (y por tanto anotando a):

- Tipos: clases, interfaces, enumerados, anotaciones.
- Constructores, métodos y campos.
- Paquetes.
- Parámetros, variables locales, variables de bucles, parámetros de *catch* y valores de enumerados.

De nuevo por simplicidad en el diseño de **Ann** se han tenido en cuenta únicamente las dos primeras opciones, dado que también constituyen las más frecuentes.

Desde Java 8, también se pueden emplear anotaciones sobre cualquier utilización de un tipo (*castings*, claúsulas *implements*, claúsulas *throws*...) [23].

2.1.3. Retención

Se distinguen tres tipos de anotaciones en función de su tiempo de retención a lo largo del programa:

- Descartadas por el compilador. Son aquellas que únicamente aparecen en el código fuente.
- Compiladas en el fichero binario correspondiente a una clase o interfaz. Dentro de este tipo encontramos a su vez dos subtipos:

- Aquellas ignoradas por la máquina virtual de Java.
- Aquellas leídas por la máquina virtual de Java cuando la clase o interfaz que las contiene se carga. Son por tanto visibles en tiempo de ejecución.

En el diseño de muchas anotaciones se establece como tipo de retención la más alta, ya que es la única que permite acceder a los datos almacenados en una anotación en tiempo de ejecución mediante la reflexión, utilizando la *Java Reflection API*.

2.1.4. Meta-anotaciones

Como tipo especial de anotaciones, existen las llamadas meta-anotaciones, que son anotaciones creadas específicamente para anotar a otras en su definición.

Las más comunes, y además relevantes en nuestra discusión, son:

- `Target`. Utilizada, como su nombre indica, para restringir el tipo de elementos que la anotación declarada anotará. Esto quiere decir que si se omite, se considera que puede anotar todos los elementos señalados en la Sección 2.1.2.

  Contiene un único miembro que es un *array* de `java.lang.ElementType`.

- `Retention`. Indica el tipo de retención a aplicar en la anotación, de acuerdo con los tipos señalados en la Sección 2.1.3. Si esta meta-anotación no aparece, se considera retención de nivel intermedio.

  Contiene como único atributo el enumerado `java.lang.RetentionPolicy`, cuyos valores son `SOURCE`, `CLASS` y `RUNTIME`, de menor a mayor nivel de retención.

2.1.5. Utilización

Las anotaciones se consideran como modificadores a la hora de utilizarlas sobre un elemento Java. Por tanto, aunque la sintaxis habitual sea la que se presenta en la Figura 2.1, podríamos perfectamente escribirlas junto al resto de modificadores; es decir, como en la Figura 2.2.

```
@Preliminary
public class TimeTravel {
...
}
```

Figura 2.1. Uso de una anotación de marcado en Java.

Sin embargo, por convención se suelen situar delante de todos los demás modificadores y en una línea superior, sobre el elemento anotado.

La sintaxis que acabamos de ver corresponde a una anotación de marcado. En el caso de anotaciones que tengan miembros, se debe especificar el valor de cada miembro que no tenga

```
public @Preliminary class TimeTravel {
...
}
```

Figura 2.2. Combinación de una anotación con modificadores.

valor por defecto a la hora de utilizar la anotación. Para asignar dichos valores se utilizan pares *identificador* = *valor* separados por coma. El orden de asignación de los miembros no importa.

En la Figura 2.3, se muestra el uso de una anotación que contiene varios miembros.

```
@RequestForEnhancement(
    id       = 2868724,
    synopsis = "Provide time-travel functionality",
    engineer = "Mr. Peabody",
    date     = "4/1/2004"
)
public static void travelThroughTime(Date destination) {
...
}
```

Figura 2.3. Uso de una anotación con varios miembros.

En aquellas anotaciones que solamente tengan un miembro, si se le asigna el identificador *value*, se puede omitir el identificador a la hora de utilizar la anotación. Por ejemplo, en la Figura 2.4 vemos el uso de la anotación `Author`, que tiene un único miembro `String`.

```
public @interface Author {
    String value();
}

@Author("Mary")
public class Person {
...
}
```

Figura 2.4. Uso de una anotación con un único miembro.

Con la nueva versión de Java se permite la utilización de una misma anotación múltiples veces sobre un mismo elemento (característica no disponible previamente) si se marca la anotación de forma especial en su definición [22].

2.1.6. Procesamiento

Java proporciona dentro del paquete `javax.annotation.processing` un conjunto específico de elementos que se centran en el procesamiento de anotaciones en tiempo de compilación.

Los procesadores de anotaciones generalmente revisan el código buscando elementos anotados con una o varias anotaciones y en función de lo que encuentren generan o no errores y *warnings* en el compilador.

La sintaxis habitual para un procesador de anotaciones es la que podemos ver en la Figura 2.5. Nótese el uso a su vez de anotaciones (`SupportedAnnotationTypes`, `SupportedSourceVersion` y `Override`).

```
@SupportedAnnotationTypes("Entity")
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class EntityProcessor extends AbstractProcessor
{
    @Override
    public boolean process(Set<? extends TypeElement> annotations,
        RoundEnvironment objects)
    {

        for (Element element : objects.getElementsAnnotatedWith(Entity.class))
        {
            // do something

            // if error
            this.processingEnv.getMessager().printMessage
            (
                Kind.ERROR,
                "The annotation @Entity is disallowed for this location. ",
                element
            );

        }

        return true;
    }
}
```

Figura 2.5. Ejemplo de un procesador de anotaciones estándar.

La anotación `SupportedAnnotationTypes` indica las anotaciones soportadas por el procesador, puediendo aceptar el caracter comodín (*). El método central de un procesador de anotaciones es `process`.

El procesamiento de anotaciones funciona por rondas: en cada ronda se puede solicitar a un procesador que procese un subconjunto de las anotaciones encontradas en el código fuente y los archivos binarios producidos en una ronda anterior. Si un procesador fue ejecutado en una ronda dada, será solicitado de nuevo en las rondas siguientes, incluida la última.

Por todo ello, el método `process` recibe como entrada dos parámetros:

- El conjunto de anotaciones que se está evaluando en la presente ronda (`annotations`).
- Elementos que se han encontrado anotados con las anotaciones que se están evaluando en la presente ronda (`objects`).

Por otro lado, el valor de retorno de un procesador de anotaciones es muy importante. Si se establece a `true` quiere decir que este procesador reclama las anotaciones que ha procesado, y por tanto ningún otro procesador se ejecutará sobre las mismas una vez el primero se haya ejecutado. Si se establece a `false` no se realiza ninguna reclamación y distintos procesadores pueden analizar el mismo tipo de anotación en busca de características distintas o para realizar acciones diferentes.

Por último, en la Figura 2.5 aparece también un ejemplo de cómo generar, de forma muy sencilla, un error en el compilador: únicamente hace falta el elemento sobre el que se quiere generar, el nivel de error y el mensaje.

2.2. Desarrollo Dirigido por Modelos

El paradigma de desarrollo que se ha seguido para la creación de **Ann** se corresponde con el denominado Desarrollo Dirigido por Modelos - *Model-Driven Development* (MDD), que utiliza los modelos como principal elemento del proceso de desarrollo. A continuación veremos los aspectos de MDD especialmente relevantes en la discusión.

La fuente bibliográfica utilizada para esta sección es el libro *Model-Driven Software Engineering in Practice* de Marco Brambilla, Jordi Cabot y Manuel Wimmer [6].

2.2.1. Modelos

Un modelo es una representación simplificada o parcial de la realidad, definida para llevar a cabo una tarea o llegar a un acuerdo sobre un tema. Tiene por tanto las siguientes dos propiedades:

- *Reducción.* Un modelo representa únicamente una selección relevante de las propiedades originales, para centrarse en los aspectos de interés.
- *Correspondencia.* Un modelo está basado en una realidad original, que se toma como prototipo y es abstraída y generalizada.

Una de las grandes ventajas de los enfoques dirigidos por modelos es que cierran la brecha de comunicación entre la fase de requisitos y análisis y la de implementación. Además es importante notar que dado que un modelo es por definición una simplificación, nunca representará de forma completa la realidad que abstrae.

2.2.2. Lenguajes de Modelado

Un lenguaje de modelado es una herramienta conceptual para describir la realidad de forma explícita, a un cierto nivel de abstracción y desde un punto de vista concreto. Se define mediante tres elementos claves:

- **Sintaxis abstracta**. Describe la estructura del lenguaje y la forma en la que los diferentes elementos se pueden combinar, independientemente de una representación o codificación en particular.
- **Sintaxis concreta**. Describe la representación específica del lenguaje de modelado, cubriendo aspectos como la codificación y/o apariencia visual. La sintaxis concreta puede ser textual o gráfica.
- **Semántica**. Describe el significado de los elementos definidos en el lenguaje y el significado de las distintas formas de combinarlos.

Una de las principales clasificaciones de los lenguajes de modelado es en función del dominio de aplicación.

Un Lenguaje de Modelado de Dominio Específico - *Domain-Specific Modeling Language* (DSL) es un lenguaje diseñado para un dominio o contexto específico, con el objetivo de facilitar la tarea de las personas que necesitan describir elementos en ese dominio.

En contrapartida, un Lenguaje de Modelado de Propósito General - *General-Purpose Modeling Language* (GPL) comprende nociones de modelado que pueden ser aplicadas a cualquier sector o dominio.

Sin embargo, la distinción entre DSL y GPL no siempre es determinista y está bien definida. Por ejemplo, *Unified Modeling Language* (UML) es un GPL que sirve para describir un sistema desde distintas perspectivas mediante la utilización de diagramas. Sin embargo, si se piensa en el problema general de modelar como un dominio específico, entonces UML puede ser visto como un DSL para la especificación de sistemas software (principalmente orientados a objetos).

2.2.3. Meta-modelos

Dado que los modelos juegan un papel central en MDD, un paso natural a continuación es representar los propios modelos como «instancias» de otros modelos más abstractos. De esta forma, del mismo modo en el que definimos un modelo como una abstracción de un fenómeno

en el mundo real, podemos definir un meta-modelo como una abstracción superior, que destaca las propiedades del modelo en si.

Un meta-modelo por tanto es la definición de la sintaxis abstracta de un lenguaje de modelado, dado que constituye una forma de describir todos los tipos de modelos que pueden ser representados mediante ese lenguaje.

Una vez visto este nivel superior, vemos que se puede iterar recursivamente para obtener infinitos niveles de meta-modelado. Sin embargo, en la práctica se ha comprobado que los meta-meta-modelos pueden ser definidos por sí mismos, y por tanto normalmente no se va más allá de este nivel de abstracción.

A cualquier nivel al que consideremos el meta-modelado, decimos que el modelo es *conforme* a su meta-modelo; de la misma forma que un programa es *conforme* a la gramática del lenguaje de programación en el que está escrito.

En la Figura 2.6 podemos ver la representación gráfica de los niveles de modelado de los que hemos hablado. Además, en la Figura 2.7 aparecen dichas capas relacionadas con los objetos que representan o definen.

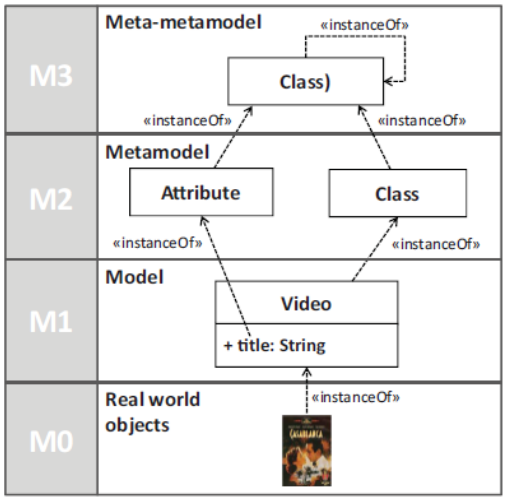

Figura 2.6. Modelos, meta-modelos y meta-meta-modelos [6].

2.2.4. Generación de código e interpretación de modelos

La semántica de un lenguaje de modelado se puede definir de varias formas:

- Definiendo todos los conceptos, propiedades, relaciones y restricciones mediante un lenguaje formal.
- Implementando generadores de código que definen implícitamente la semántica del lenguaje a través del código generado.

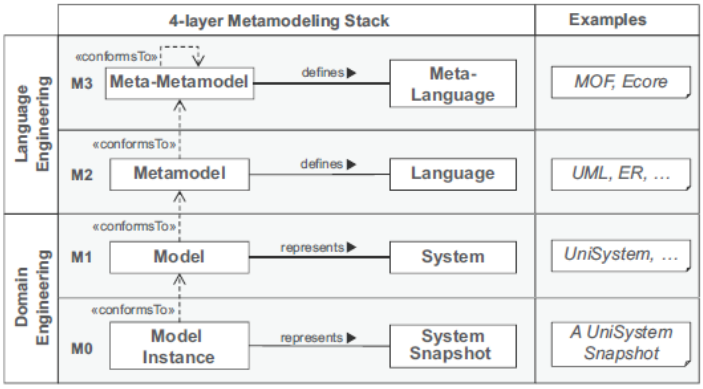

Figura 2.7. Las cuatro capas del meta-modelado [6].

- Definiendo transformaciones para simular el comportamiento del modelo.

Por otro lado, un modelo es lo suficientemente completo como para ser ejecutable si, desde un punto de vista teórico, su semántica operacional está completamente especificada.

En la práctica, la ejecutabilidad del modelo puede depender más del motor de ejecución seleccionado que del modelo en sí. Para hacer por tanto que los modelos ejecutables se ejecuten de forma efectiva, se utilizan dos estrategias alternativas: la generación de código y la interpretación de modelos.

Un generador de código se puede pensar como un «compilador de modelos», que genera código ejecutable desde un modelo de alto nivel para crear una aplicación funcional. Esta generación se suele realizar utilizando motores de plantillas basados en reglas; es decir, conjuntos de plantillas con huecos que se rellenarán una vez instanciados los elementos del modelo.

En contrapartida, la interpretación de modelos se basa en implementar un motor genérico que traduce y ejecuta el modelo sobre la marcha, con exactamente el mismo enfoque de los intérpretes para los lenguajes de programación interpretados.

2.3. Eclipse Modeling Framework

Eclipse es un Entorno de Desarrollo Integrado - *Integrated Development Environment* (IDE) que contiene un sistema de *plug-in*s para su extensión y personalización.

Ann se ha desarrollado utilizando *Eclipse Modeling* Framework (EMF), una de las iniciativas MDD más conocidas. Aunque dentro de EMF se pueden encontrar muchas facilidades para el MDD en forma de componentes o *plug-in*s, nos centraremos en los que se han utilizado a lo largo del proyecto.

De nuevo la fuente utilizada principalmente pare esta sección ha sido el libro *Model-Driven Software Engineering in Practice* de Marco Brambilla, Jordi Cabot y Manuel Wimmer [6], indicándose explícitamente cuando no sea así.

2.3.1. Ecore

Ecore es el lenguaje para la definición de meta-modelos en EMF. Está basado en un subconjunto de diagramas de clase UML para la descripción de aspectos estructurales, y adaptado a Java a propósito de su implementación.

En la Figura 2.8 se pueden ver jerarquizados los distintos componentes de Ecore.

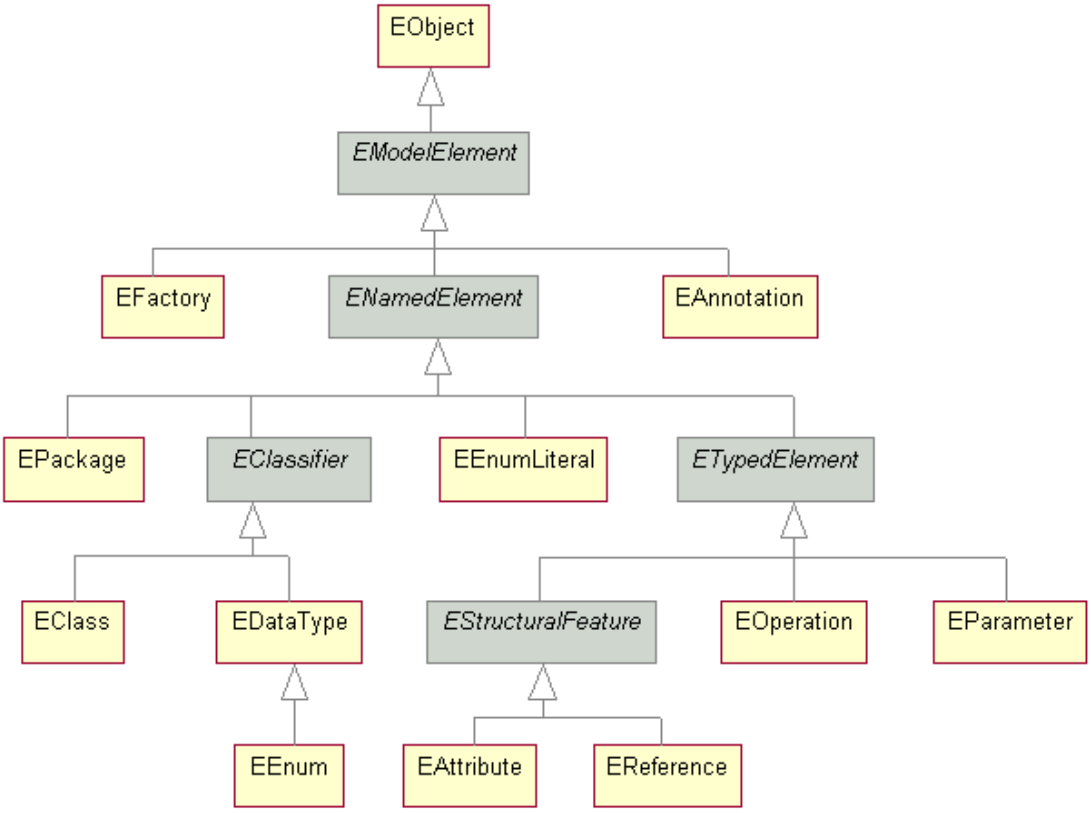

Figura 2.8. Meta-modelo simplificado de Ecore [24].

Para describir meta-modelos basados en Ecore, existen diversas sintaxis concretas, tanto gráficas como textuales, soportadas por distintos editores. En la descripción de la sintaxis abstracta de **Ann** se ha utilizado un editor textual de Ecore, Emfatic, para mayor comodidad, al ser el editor nativo de EMF en forma de árbol.

Emfatic proporciona una sintaxis similar a Java y además permite generar su código a partir de meta-modelos Ecore ya existentes. Proporciona además la opción de actualizar el meta-modelo Ecore cada vez que se guarde el archivo Emfatic, olvidándose así el usuario de estar utilizando un editor ajeno.

2.3.2. Xtext

Para el desarrollo de la sintaxis concreta de **Ann** se ha utilizado Xtext, un *framework* de código libre especializado en sintaxis textuales.

Xtext presenta la gran ventaja de que además de generar un analizador sintáctico como la mayoría de entornos de desarrollo de lenguajes textuales, también genera un IDE completo y personalizable basado en Eclipse para el lenguaje definido. Las funcionalidades que el IDE generado proporciona incluyen: resaltado de sintaxis, vista en árbol, navegador de código fuente, análisis estático, etc.

2.3.3. Xtend

Xtend es un dialecto de Java más expresivo y flexible [25]. Se centra en proporcionar una sintaxis más concisa y funcionalidad adicional a la que ofrece Java, como inferencia de tipos o algunas características de la programación funcional. Además, permite el desarrollo de plantillas, lo que lo hace ideal para la generación de código en un DSL.

Es el lenguaje que se ha utilizado, encuadrado dentro del *framework* de Xtext, para definir la semántica de **Ann**.

3. Trabajo relacionado

El campo del diseño de anotaciones Java está bastante inexplorado hasta la fecha en cuanto al análisis del meta-modelo subyacente se refiere. Sí que se pueden encontrar más fácilmente artículos que expanden o utilizan el soporte sintáctico actual como son:

- Viera K. Proulx y Weston Jossey proporcionan una librería basada en las anotaciones de Java y la reflexión para ayudar a los programadores «novatos» en el desarrollo de pruebas unitarias [11].
- Glauber Ferreira, Emerson Loureiro y Elthon Oliveira analizan los sistemas de software para la verificación de modelos desde una nueva perspectiva en la que, en vez de inferirse el modelo del código Java y analizarse posteriormente, se verifica en tiempo de compilación mediante el uso de anotaciones [12].
- Walter Cazzola y Edoardo Vacchi proponen una extensión a Java, que llaman *@Java*, para expandir el ámbito de aplicación actual de las anotaciones a bloques de código y expresiones [13].

Además de los ejemplos mostrados anteriormente, existen otros artículos en los que sí se ha analizado el meta-modelo:

- Andrew Phillips intenta conciliar el diseño actual de anotaciones con los principios de la programación orientada a objetos mediante la introducción de una nueva «macro-anotación» (como él mismo la denomina): `composite`. Gracias a ella consigue dar soporte a la composición, permitiendo así la encapsulación y el polimorfismo de anotaciones [14].
- Federico Mancini, Dag Hovland y Khalid A. Mughal analizan las anotaciones Java cuando se aplican a pruebas de validación de entradas, enumerando las limitaciones que encuentran [7].

Los casos anteriores no los analizaremos más a fondo porque si bien se centran en el meta-modelo de las anotaciones, lo hacen en contextos específicos, como es la programación orientada a objetos en el primero y las pruebas de validación en el segundo.

Por último, merece la pena mencionar a Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae y James Noble, que gracias a su conjunto de herramientas *JavaCOP* proporcionan un medio para verificar restricciones de un programa Java, basándose

en su árbol de sintaxis [9]. Si bien esto soluciona una parte de lo mencionado en la Sección 1.1, su objetivo directo no son las anotaciones Java, sino proporcionar un sistema completo de verificación de restricciones en Java.

Al margen de todos estos desarrollos se encuentran otros dos que si es necesario que examinemos con más detalle, por ser sus objetivos muy cercanos a los que persigue **Ann**.

En primer lugar, Carlos Noguera y Renaud Pawlak proponen AVal [8], un conjunto de meta-anotaciones para añadir restricciones a las ya definidas. Exploraremos esta propuesta más en detalle en la Sección 3.1.

Por otro lado, Ian Darwin [10] proporciona un sistema muy similar a AVal pero que en vez de basarse en meta-anotaciones y ligar las restricciones a la definición de una anotación utiliza una sintaxis de reclamos o aserciones (*claims*) sobre anotaciones ya existentes. Analizaremos este sistema en la Sección 3.2.

3.1. AVal

Como hemos visto en los capítulos 1 y 2, aunque hay algo de soporte en Java actualmente para la expresión de restricciones sobre conjuntos de anotaciones, frecuentemente no es suficiente. En este contexto surge el validador de anotaciones AVal, defendiendo la idea de que una anotación debería contener en su definición la información acerca de cómo ha de ser validada.

AVal proporciona un conjunto de meta-anotaciones para la expresión de restricciones, que en adelante llamaremos `@Validadores`. Esta idea surge a raíz de la meta-anotación `Target` proporcionada por Java, que efectivamente ya sirve para describir, aunque de forma escasa, un requisito.

La arquitectura de AVal se compone de cuatro capas, como vemos en la Figura 3.1, donde la nomenclatura `@DSL` se utiliza para englobar a las anotaciones específicas del dominio. Cada `@Validador` representa una regla de validación y está anotado él mismo con la clase que la implementa y que es la encargada en última instancia de realizar la validación, apoyándose en los procesadores Spoon [29].

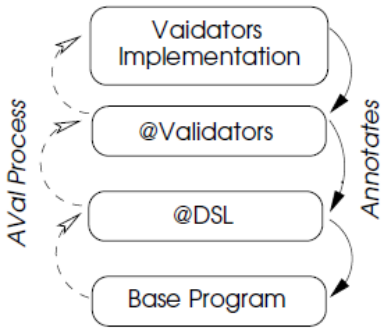

Figura 3.1. Arquitectura de AVal en cuatro capas y sus interacciones [8].

AVal se ejecuta antes de la fase de generación de código o compilación: recorre el código fuente buscando anotaciones de dominio específico, y cada vez que encuentra un elemento anotado, ejecuta cada una de las implementaciones de los `@Validadores` asociados a la anotación.

3.1.1. @Validadores genéricos

AVal proporciona varios `@Validadores` útiles en todos los dominios (además de permitir crearlos personalizados), que subdivide en dos tipos: estructurales y de valor.

Dentro de los `@Validadores` estructurales encontramos:

- `@AVAlTarget`. Proporciona una extension de la anotacion `Target` de Java, pero más específica. Actualmente, si se quiere utilizar una anotación exclusivamente en interfaces, se tendrá que utilizar `@Target(ElementType.TYPE)`. Sin embargo, esto permite anotaciones, enumerados y clases; por lo que habría que realizar un procesador o similar que comprobase por otros medios que efectivamente se están anotando únicamente interfaces, o arriesgarse al mal uso de la anotación y relegarlo a una observación en un comentario.
- `@Inside`. Especifica que el contenedor del elemento que esté anotado con la presente anotación tiene que estar anotado con la anotación que se dé como parámetro.
- `@Prohibits`. Los elementos anotados con la anotación actual no pueden estar anotados con la anotación dada como parámetro.
- `@Requires`. Es el opuesto a `@Prohibits`.

Por otro lado, los `@Validadores` de valor comprenden:

- `@RefersTo`. Indica que el valor del atributo anotado con esta anotación debe ser igual que el de otro atributo de otra anotación presente en el programa.
- `@Matches`. Se aplica a atributos de tipo String y comprueba que los valores del atributo se corresponden con una expresión regular Java dada.
- `@Unique`. Comprueba que un valor de atributo dado es único dentro del conjunto de anotaciones del mismo tipo en el programa.

3.1.2. Análisis

AVal constituye una propuesta sólida y con diversos ámbitos de aplicación, como podemos extraer analizando simplemente los `@Validadores` que proporciona. Además, se integra muy bien en el contexto actual de definición de anotaciones en Java, utilizando la misma filosofía de meta-anotaciones para las restricciones que ya dejaba entrever la anotación nativa `Target`.

Por otro lado, hay que tener en cuenta que el uso de AVal sería poco atractivo en el caso de grandes *framework*s con un uso intensivo de anotaciones, dado que requiere una redefinición

y revisión de todas las que se utilicen para poder adaptarlo. Ésta es la mayor desventaja de AVal, puesto que, como señalan los propios autores, su motivación es aplicarlo a *frameworks* orientados a atributos, que son justamente los que más van a utilizar las anotaciones.

3.2. AnnaBot

AnnaBot es también un validador de anotaciones, en el que las restricciones asociadas a una anotación se describen mediante *claims*, es decir, aserciones sobre lo que queremos que cumpla un conjunto de anotaciones. Consiste en un DSL para describir dichas aserciones y un procedimiento basado en la reflexión para comprobar que se cumplen.

Un ejemplo de su sintaxis concreta lo encontramos en la Figura 3.2. Aparece una aserción en la que se describen varias restricciones JPA; veamos las dos primeras. Una clase anotada con `Entity` debe contener la anotación `Id`, que marca la clave primaria de la entidad de la base de datos que se quiere modelar. Sin embargo, en una entidad JPA, o bien los campos o bien los métodos pueden estar anotados con anotaciones de persistencia, pero no ambos simultáneamente. Ninguna de estas dos características se pueden expresar mediante el soporte actual de Java.

```
import javax.persistence.Entity;
import javax.persistence.Id;

claim JPA {
if (class.annotated(javax.persistence.Entity)) {
    require method.annotated(javax.persistence.Id)
        || field.annotated(javax.persistence.Id);
  atMostOne     method.annotated(javax.persistence.ANY)
||      field.annotated(javax.persistence.ANY)
        error "The JPA Spec only allows JPA annotations on methods OR fields";
};
if (class.annotated(javax.persistence.Embeddable)) {
        noneof   method.annotated(javax.persistence.Id) ||
        field.annotated(javax.persistence.Id);
};
}
```

Figura 3.2. Ejemplo de sintaxis concreta de AnnaBot [10], con una aserción para JPA.

La sintaxis concreta de AnnaBot es muy similar a Java, y el autor manifiesta que esto es así intencionadamente. Aunque se presenta dicha sintaxis, el DSL no está completo, ya que el autor señala que en el momento de la escritura del artículo el generador de código no se encuentra todavía implementado. Por tanto, las aserciones únicamente se pueden escribir en la actualidad en código Java directamente.

En la Figura 3.3 se muestra la traducción a Java de la segunda restricción explicada anteriormente.

AnnaBot es un proyecto de código abierto que todavía está creciendo y al cuál los usuarios pueden contribuir añadiendo nuevos tipos de aserciones.

```
package jpa;

import annabot.Claim;
import tree.*;

public class JPAEntityMethodFieldClaim extends Claim
{
        public String getDescription() {
                return "JPA Entities may have field
OR method annotations, not both";
        }
        public Operator[] getClassFilter() {
                return new Operator[] {
                new ClassAnnotated(
                "javax.persistence.Entity"),
                };
        }

        public Operator[] getOperators() {
                return new Operator[] {
                    new AtMostOne(
                    new FieldAnnotated(
                        "javax.persistence.*"),
                    new MethodAnnotated(
                        "javax.persistence.*",))
                };
        }
}
```

Figura 3.3. Aserción de AnnaBot expresada en código Java como una clase.

3.2.1. Análisis

Una desventaja de AnnaBot consiste en que aunque se proporcione la sintaxis concreta del DSL, éste se encuentra totalmente inoperativo en la actualidad. Además, como podemos ver en la Figura 3.4, las aserciones que se pueden declarar no contemplan las restricciones que una anotación pueda conllevar en sí misma; como por ejemplo, la restricción que comentábamos en la introducción de que `Entity` no puede tener como *target* una clase final. Sin embargo, según el autor este tipo de aserciones serían fácilmente incorporables a la funcionalidad actual de AnnaBot.

Por otro lado, todas las anotaciones a procesar deben encontrarse definidas con la meta-anotación `@Retention(RetentionPolicy.RUNTIME)` debido a que utiliza la reflexión para realizar las comprobaciones.

Por último, todas las verificaciones que actualmente posibilita la sintaxis se podrían realizar en tiempo de compilación utilizando procesadores de anotaciones, sin necesidad de recurrir a la *Java Reflection API* que la propia documentación de Java aconseja evitar siempre que se disponga de alternativas [21].

Esto es así debido a que la reflexión trae consigo varias desventajas:

- Una sobrecarga en el rendimiento.
- Los permisos que requiere, de ejecución, no tienen por qué estar garantizados.

```
program:      import_stmt*
        CLAIM IDENTIFIER '{'
              stmt+
        '}'
        ;

import_stmt:   IMPORT NAMEINPACKAGE ';'
        ;

// Statement, with or without an
// if... { stmt } around.
stmt:   IF '(' checks ')' '{' phrase+ '}' ';'
        | phrase
        ;
phrase: verb checks error? ';'
        ;

verb:   REQUIRE | ATMOSTONE | NONEOF;

checks:  check
               | NOT check
               | ( check OR check )
               | ( check AND check )
               | ( check ',' check)
               ;

check:  classAnnotated
        | methodAnnotated
        | fieldAnnotated;

classAnnotated: CLASS_ANNOTATED '('
         NAMEINPACKAGE ')';
methodAnnotated:        METHOD_ANNOTATED '('
         NAMEINPACKAGE ')';
fieldAnnotated: FIELD_ANNOTATED '('
         NAMEINPACKAGE
        ( ',' MEMBERNAME )? ')'
        ;

error:  ERROR QSTRING;
```

Figura 3.4. Sintaxis inicial del DSL AnnaBot.

- Rompe las abstracciones al permitir operaciones que serían ilegales en código no reflexivo, como por ejemplo el acceso a campos privados, pudiendo llevar a comportamientos inesperados y código no portable

En cuanto a aportaciones de AnnaBot, es importante destacar que permite expresar restricciones sobre conjuntos de anotaciones ya definidos sin necesidad de volver a crearlos, al contrario de lo que sucedía con AVal.

4. Diseño

Como ya hemos visto en capítulos anteriores, para suplir algunas de las carencias del soporte sintáctico de Java para el diseño de anotaciones se decidió crear un Lenguaje de Modelado de Dominio Específico.

El criterio de decisión para elegir un lenguaje de modelado ha sido que el principal objetivo de **Ann** es hacer explícita una mayor parte del modelo conceptual que hay detrás de un conjunto de anotaciones. Ésta es precisamente la tarea para la que se utilizan los lenguajes de modelado, como vimos en el Capítulo 2: son herramientas para describir la realidad de forma explícita, a un cierto nivel de abstracción y desde un punto de vista en concreto.

Además, para permitir una mayor capacidad expresiva y simplicidad se ha optado por restringir el lenguaje de modelado al dominio del diseño de anotaciones.

En este apartado por tanto se estudiará la arquitectura de la solución así como el diseño específico de cada uno de los componentes del DSL **Ann**.

4.1. Arquitectura

Como todo lenguaje de modelado, **Ann** consta de tres componentes principales:

- **Sintaxis abstracta**, definida por un meta-modelo de las anotaciones Java.
- **Sintaxis concreta**, textual en este caso.
- **Semántica**, implícita en un generador de código Java.

Los criterios de decisión y diseño para cada uno de los componentes anteriores serán estudiados en las secciones siguientes.

En la Figura 4.1 se muestra representada gráficamente la arquitectura de la solución .

Vemos que en apoyo al DSL han aparecido un editor y un validador de la coherencia de las instancias del meta-modelo del DSL.

4.2. Meta-modelo

Para describir la sintaxis abstracta de **Ann** se ha definido un meta-modelo que representa los distintos componentes del diseño de anotaciones y las relaciones entre ellos (como ya vimos en el Capítulo 2, los meta-modelos son sintaxis abstractas de lenguajes de modelado).

En nuestro meta-modelo, una anotación se compone de los siguientes elementos:

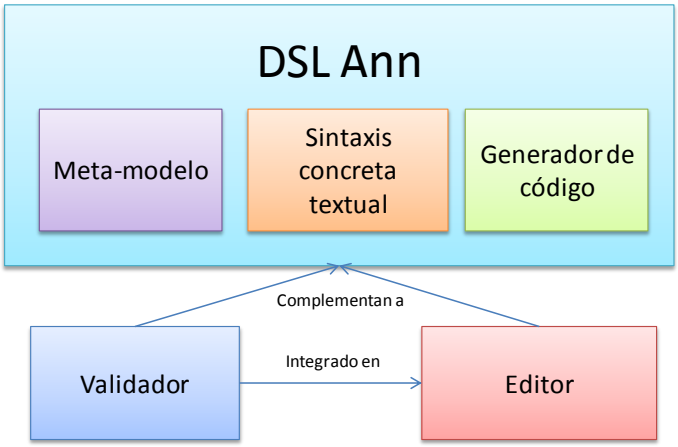

Figura 4.1. Arquitectura de la solución.

- Nombre
- Tipo de retención. Como vimos en el Capítulo 2, existen tres tipos: de código fuente, de clase y de tiempo de ejecución.
- Atributos o campos de la anotación. En la Sección 4.2.2 se verá en detalle su meta-modelo.
- Un conjunto de restricciones que afectan a la anotación. Dichas restricciones se verán en detalle en el siguiente apartado.

En la Figura 4.2 podemos ver la representación gráfica del meta-modelo explicado.

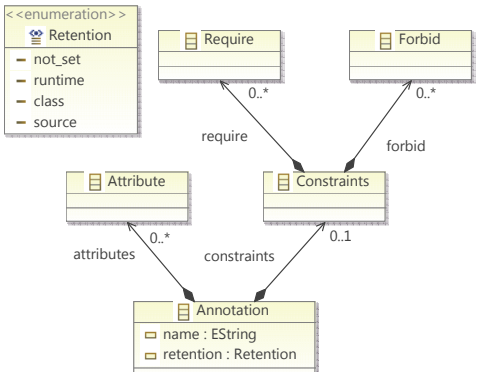

Figura 4.2. Meta-modelo de anotaciones simplificado.

4.2.1. Restricciones

El meta-modelo de restricciones constituye una de las características clave de **Ann**.

Actualmente existen dos tipos de restricciones: `Require` o requisitos y `Forbid` o prohibiciones. Ambos pueden estar centrados en un *target* en concreto; en dicho caso, la restricción se referirá a características que debe cumplir el *target* indicado.

Para expresar las restricciones se utilizan *statement* o sentencias, compuestas de un tipo de elemento Java, modificadores y una anotación, pudiendo combinarlos de forma opcional. Se han seleccionado para el meta-modelo los modificadores más comúnmente utilizados: aquellos de visibilidad, `static`, `final` y `abstract`.

En un `Require`, la semántica de un conjunto de sentencias es que al menos una de ellas se tiene que cumplir. En un `Forbid`, lo prohibido es la simultaneidad de lo descrito por las sentencias.

Por último, dentro de los `Require` centrados en un *target* contenedor se puede indicar que todos los elementos que sean del tipo señalado en los *statement* lo cumplan; por ejemplo, si queremos que todos los métodos de una clase anotada con una anotación sean públicos.

Con todo esto se dispone de muchas variantes de restricciones (`Require` o `Forbid`, con o sin *target*, con uno o varios *statement* o sentencias, etc.); sin embargo no todas ellas son coherentes. Es ahí donde interviene la validación; y veremos cómo se ha llevado a cabo en el Capítulo 5.

En la Figura 4.3 encontramos la representación gráfica del meta-modelo descrito.

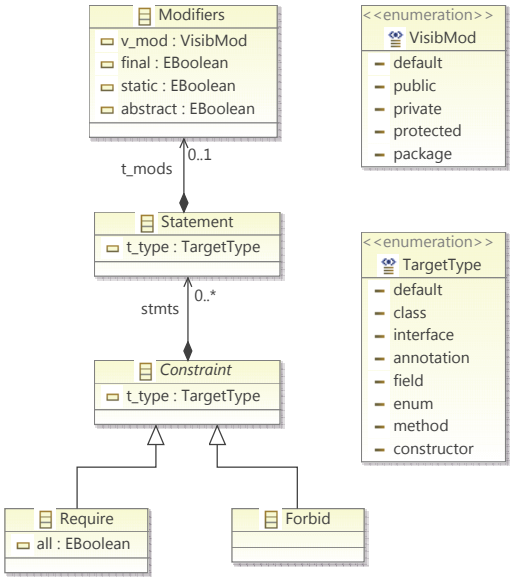

Figura 4.3. Meta-modelo de restricciones.

4.2.2. Atributos

Finalmente, el meta-modelo de atributos es el más sencillo. Describe los posibles miembros que puede tener una anotación, atendiendo a lo que se explicó en el Capítulo 2. Cada miembro

tiene a su vez como atributo un posible valor por defecto, del tipo de dato que es el miembro (nótese que los tipos de datos de Ecore se corresponden con los de Java [24]).

En la Figura 4.4 encontramos la representación gráfica del meta-modelo de atributos.

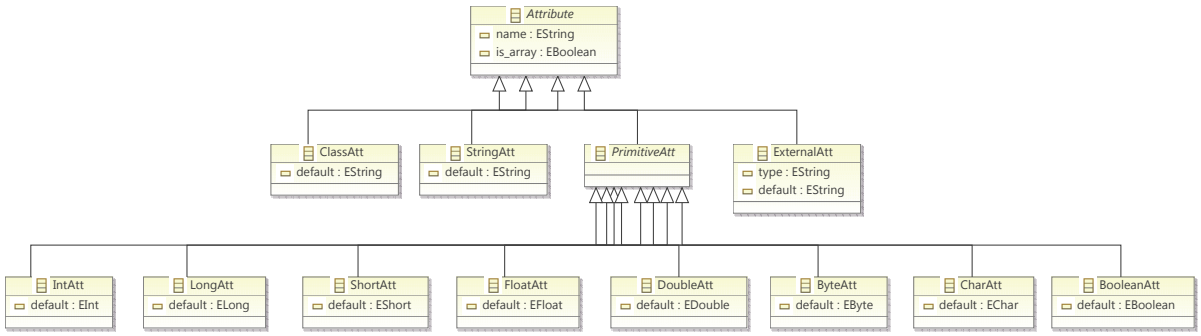

Figura 4.4. Meta-modelo de atributos de una anotación.

Cabe destacar que `ExternalAtt` representa atributos que pueden estar declarados de forma externa al DSL; y por tanto engloba anotaciones y enumerados.

Por último, el valor booleano `is_array` representa, como su propio nombre indica, si el atributo es simple o constituye un *array* de atributos del mismo tipo.

4.3. Sintaxis concreta

Se ha decidido utilizar una sintaxis concreta textual para el DSL **Ann** debido por un lado a que los usuarios objetivos del mismo son programadores en Java; y por otro a que uno de sus objetivos es encajar mejor dentro de la sintaxis general de Java para contenedores, y esto no es posible si se utiliza una sintaxis concreta gráfica, no presente en dicho lenguaje.

Dicha sintaxis concreta se puede encontrar en su versión completa en el Anexo A, descrita en *Extended Backus–Naur Form* (EBNF). En esta sección señalaremos únicamente algunos puntos importantes.

El esquema general de sintaxis para una anotación se puede ver en la Figura 4.5.

⟨*Annotation*⟩ ::= ⟨*Retention*⟩? '`annotation`' ⟨*ID*⟩ '`{`' (⟨*Attribute*⟩ '`;`')* ⟨*Constraints*⟩? '`}`';

⟨*Constraints*⟩ ::= (⟨*Require*⟩ | ⟨*Forbid*⟩)+;

Figura 4.5. Fragmento de sintaxis de anotaciones.

Vemos que se trata de un esquema sencillo, en el que se especifica primero la información general (tipo de retención e identificador) y posteriormente los atributos y restricciones, cuya

sintaxis particular veremos a continuación.

4.3.1. Atributos

Gracias a la Figura 4.6 podemos comprobar cómo la sintaxis para definir un atributo de una anotación se ha definido acorde a la sintaxis habitual de Java. En concreto:

- Los atributos ya no comparten la sintaxis de declaración de métodos, sino la de campos; es decir, se utiliza un identificador en lugar de un identificador seguido de dos paréntesis.
- Para asignar el valor por defecto, se utiliza el caracter «=» y no la palabra clave *default*.

⟨*LongAtt*⟩ ::= '`long`' ('`[]`')? ⟨*ID*⟩ ('`=`' ⟨*INT*⟩)?;

⟨*ShortAtt*⟩ ::= '`short`' ('`[]`')? ⟨*ID*⟩ ('`=`' ⟨*INT*⟩)?;

⟨*FloatAtt*⟩ ::= '`float`' ('`[]`')? ⟨*ID*⟩ ('`=`' ⟨*FLOAT*⟩)?;

Figura 4.6. Fragmento de sintaxis de atributos.

4.3.2. Restricciones

Veamos ahora la sintaxis correspondiente a las restricciones en la Figura 4.7.

⟨*Forbid*⟩ ::= '`forbid`' ⟨*Statement*⟩ ('`and`' ⟨*Statement*⟩)* '`;`'
| '`at`' ⟨*TargetType*⟩ '`:`' '`forbid`' ⟨*Statement*⟩ ('`and`' ⟨*Statement*⟩)* '`;`';

⟨*Require*⟩ ::= '`require`' ⟨*Statement*⟩ ('`or`' ⟨*Statement*⟩)* '`;`'
| '`at`' ⟨*TargetType*⟩ '`:`' '`require`' '`all`'? ⟨*Statement*⟩ ('`or`' ⟨*Statement*⟩)*
'`;`';

⟨*Statement*⟩ ::= ⟨*AnnID*⟩
| ⟨*TgtStatement*⟩;

⟨*TgtStatement*⟩ ::= ⟨*AnnID*⟩? ⟨*Modifiers*⟩ ⟨*TargetType*⟩;

Figura 4.7. Fragmento de sintaxis de restricciones.

Recordemos que una característica del meta-modelo de restricciones era que se permitía la multiplicidad de sentencias o *statements* en una restricción, teniendo un significado distinto en función del tipo del que se tratase. Observamos que la sintaxis refleja este hecho mediante las palabras clave `or` y `and` que sirven de conectores entre las distintas sentencias de los `Require` y `Forbid`, respectivamente.

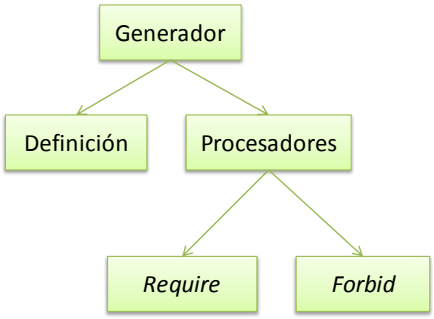

Figura 4.8. Estructura del generador de código.

4.4. Generador de código

Para especificar la semántica de un DSL ya vimos en el Capítulo 2 que había dos alternativas: la generación de código o la interpretación del modelo. Las ventajas de la primera respecto a la segunda son [6]:

- El código generado está producido en un lenguaje de programación estándar que cualquier desarrollador puede entender.
- Permite a los usuarios elegir el entorno de ejecución que prefieran. Con esto se da mayor flexibilidad al no depender de la tecnología MDD utilizada, pudiendo incluso migrar u optar en el futuro por abandonarla al producirse código estándar.
- Permite reutilizar objetos de programación ya existentes. En particular, trozos de código ya existentes se pueden generalizar y usar como plantillas para la generación de nuevas piezas de software. Si el generador de código es lo suficientemente flexible se puede incluso extender de forma iterativa para generar piezas de código cada vez más ricas.
- Un generador de código suele ser más fácil de mantener, depurar y revisar al consistir típicamente en transformaciones basadas en reglas, mientras que un intérprete tiene un comportamiento genérico y complejo al tener que cubrir todos los casos posibles de ejecución.
- En general, una aplicación generada tiene mejor rendimiento en términos de velocidad de ejecución que la correspondiente versión interpretada.

Dado que todas las características anteriores sintonizan especialmente bien con los objetivos perseguidos por **Ann**, se ha decidido utilizar un generador de código para describir su semántica.

En la Figura 4.8 podemos ver la representación gráfica de la estructura del generador de código.

Es necesario generar por un lado el código Java correspondiente a las anotaciones definidas y por otro algún mecanismo de comprobación de las restricciones asociadas a las mismas. El

mecanismo elegido para esto último han sido los procesadores de anotaciones explicados en el Capítulo 2. Esto es así debido a que todas las restricciones que se pueden modelar mediante **Ann** son verificables en tiempo de compilación, y por tanto no es necesario recurrir a la reflexión, que ya vimos en el Capítulo 3 que era mejor evitar si se disponía de alternativas.

5. Implementación

En este capítulo se estudiará la implementación del diseño propuesto en el Capítulo 4.

Como ya se mencionó en la Sección 2.3, **Ann** se ha desarrollado utilizando EMF. Dentro de dicho *framework*, han intervenido distintos componentes en la implementación de cada uno de los elementos que constituyen **Ann**; todos ellos ya explicados en la Sección 2.3:

- Para la creación de los meta-modelos se ha utilizado Emfatic, editor textual para Ecore.
- En el desarrollo de una sintaxis concreta para la gramática del DSL se ha empleado el *framework* Xtext.
- Por último, el generador de código y el validador se han implementado utilizando el lenguaje Xtend; integrado dentro de las facilidades que ofrece Xtext para la personalización del editor del DSL.

La implementación se ha realizado conforme al diseño explicado en el Capítulo 4. En las siguientes secciones se analizarán los aspectos concretos de la misma que no se infieren de forma directa a partir del diseño, ya sea por constituir una especialización del mismo o por tener en cuenta detalles adicionales. A lo largo de dicha exposición únicamente aparecerán fragmentos cuando sea necesario ilustrar un hecho, por lo que en el Anexo B se han recogido muchos más, representativos de cada parte de **Ann**.

Por último, al final del capítulo encontraremos algunos ejemplos de utilización de la sintaxis descrita; y una característica adicional: los *quickfixes*.

5.1. Meta-modelo

El editor textual Emfatic proporciona una sintaxis similar a Java para la descripción de meta-modelos Ecore. Una vez que se dispone de un meta-modelo Ecore, se pueden generar los elementos en código Java asociados al mismo. Con dichos elementos, que constituyen en su mayoría interfaces e implementaciones de las mismas, es con los que interactúan el resto de componentes de **Ann**.

La implementación de todos los meta-modelos se corresponde de forma literal con el diseño, por lo que no es necesario realizar ninguna puntualización.

5.2. Sintaxis concreta

Para la implementación de la sintaxis concreta textual se ha utilizado, como ya se ha dicho previamente, el *framework* Xtext. Cada regla en Xtext devuelve un objeto; por lo que una variación de la sintaxis de Xtext respecto a la presentada en el capítulo anterior son las asignaciones, con las particularidades de los símbolos `+=` para añadir elementos a un conjunto y `?=` para asignar valores booleanos.

En la Figura 5.1 podemos ver cómo la sintaxis de `Require` y `Forbid` se han dividido para tener en cuenta algunos requisitos de coherencia del meta-modelo, que se pueden controlar muy fácilmente y de forma natural con la sintaxis.

```
Forbid returns Forbid:
    "forbid" stmts+=Statement ("and" stmts+=Statement)* ";" |
    "at" t_type=ContainerType ":"
        "forbid" stmts+=InnerStatement ("and" stmts+=InnerStatement)* ";" |
    "at" t_type=InnerType ":"
        "forbid" stmts+=ContainerStatement ";"
;

Require returns Require:
    "require" stmts+=Statement ("or" stmts+=Statement)* ";" |
    "at" t_type=ContainerType ":"
        "require" (all?="all")? stmts+=InnerStatement ("or" stmts+=InnerStatement)* ";" |
    "at" t_type=InnerType ":"
        "require" stmts+=ContainerStatement ("or" stmts+=ContainerStatement)* ";"
;
```

Figura 5.1. Fragmento de sintaxis de restricciones.

Una cuestión importante es que tanto los `Require` como los `Forbid` que sean sobre un *target* concreto, al referirse a condiciones acerca de dicho elemento, tienen que llevar emparejados los elementos contenedores y los elementos contenidos. Es decir, si la restricción es sobre un *target* contenedor, entonces sus *statements* deben ser acerca de los elementos Java no contenedores que forman parte de él; y viceversa en caso de que la restricción sea sobre un *target* no contenedor, ya que en este caso se describirá una restricción sobre el contenedor Java del que forma parte.

Los dos tipos de sentencia por tanto asociados a elementos contenedores y no contenedores son los que aparecen en la Figura 5.1 como `ContainerStatement` y `InnerStatement`, respectivamente; así como los tipos de *target*s asociados `ContainerType` y `InnerType`. De nuevo aquí se ha tomado una simplificación al no considerar contenedores anidados.

Por último destacar también que para cada tipo Java se ha creado una regla de *statement* distinto; esto es así porque de nuevo mediante la sintaxis se controla muy fácilmente qué modifi-

cadores son aplicables a cada elemento, y por tanto qué *statements* son correctos en el contexto Java.

5.3. Generador de código

El generador de código se ha implementado siguiendo el diseño de forma literal.

Recordemos que existían tres generadores de código: el de las definiciones Java de las anotaciones, el del procesador de `Require` y el del procesador de `Forbid`. Todos ellos siguen un procedimiento muy similar: recorren la instancia del meta-modelo que ha definido el usuario mediante la sintaxis concreta del DSL utilizando el editor, y en función de las características de cada anotación que encuentren generan el código pertinente.

Cabe destacar que se ha utilizado de forma clave la facilidad de plantillas o *templates* de Xtend. El generador de la definición Java de una anotación constituye un ejemplo sencillo de esta facilidad, y se puede ver en la Figura 5.2.

```
45    def compile(Annotation a) '''
46        package «model.package»;
47
48        «IF a.retention != Retention.NOT_SET»
49        import java.lang.annotation.Retention;
50        import java.lang.annotation.RetentionPolicy;
51
52        @Retention(RetentionPolicy.«a.retention.name.toUpperCase»)
53        «ENDIF»
54        public @interface «a.name» {
55            «FOR f: a.attributes»
56            «f.compile»
57            «ENDFOR»
58        }
59    '''
60
```

Figura 5.2. Generación de la definición de una anotación.

Vemos que dentro del *template* se invoca un `compile` para cada atributo; éste da un *template* resultado de procesar el atributo de la anotación dado. En el Anexo B se verá que ésta es una práctica recurrente en el desarrollo de los generadores. Así mismo en dicho anexo aparecen, entre otros, fragmentos más extensos correspondientes a los procesadores de anotaciones.

5.4. Editor: ejemplos de sintaxis

El editor para el lenguaje es el generado por el *framework* de Xtext.

A continuación veremos dos ejemplos de utilización en la declaración de anotaciones sencillas.

En la Figura 5.3 vemos la anotación `Person` y la anotación `Version`. Además también observamos el contexto completo del editor, con el árbol de navegación del proyecto a la izquierda y

un *look & feel* similar al propio Eclipse, como comentamos en la Sección 2.3.

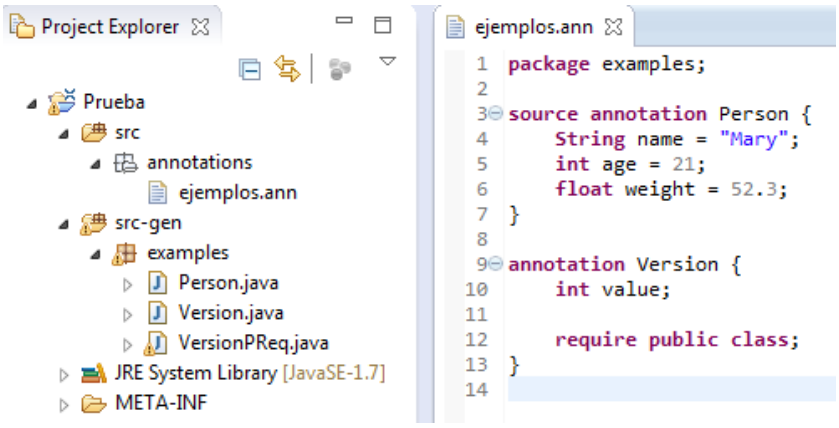

Figura 5.3. Definición de anotaciones en el editor de **Ann**.

Por otro lado, en las Figuras 5.4 y 5.5 observamos el código Java que se ha generado tras la definición de las anotaciones en la sintaxis de **Ann**.

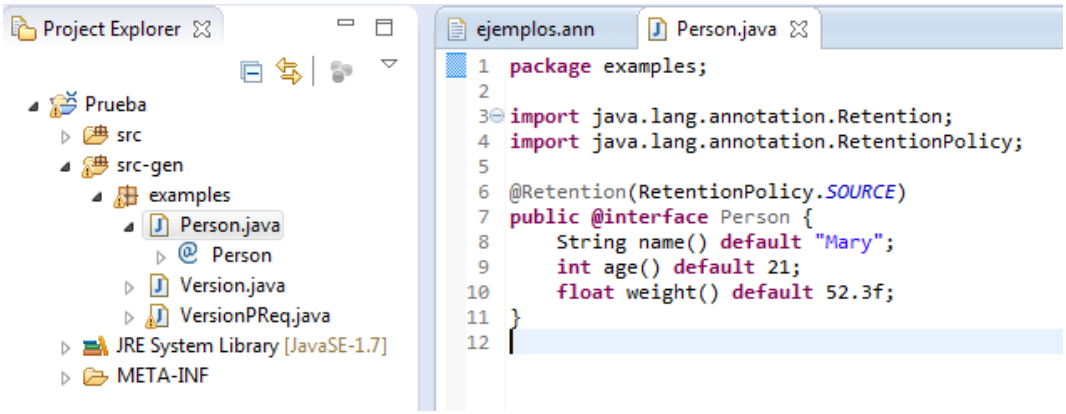

Figura 5.4. Código Java generado tras la definición de `Person`.

Además, como la anotación `Version` tenía un requisito, se ha generado el correspondiente procesador de `Require`, cuyo método `process` podemos ver en la Figura 5.6. Vemos que se realiza la comprobación expresada mediante el `Require` al definir la anotación, y además se notifica el error concreto mediante un mensaje.

5.5. Validador: notificación y *quickfix* de errores

Por último, como característica adicional del editor se proporciona un validador. Es el encargado de realizar las comprobaciones adicionales para garantizar la coherencia de los modelos obtenidos, como ya explicamos en el Capítulo 4.

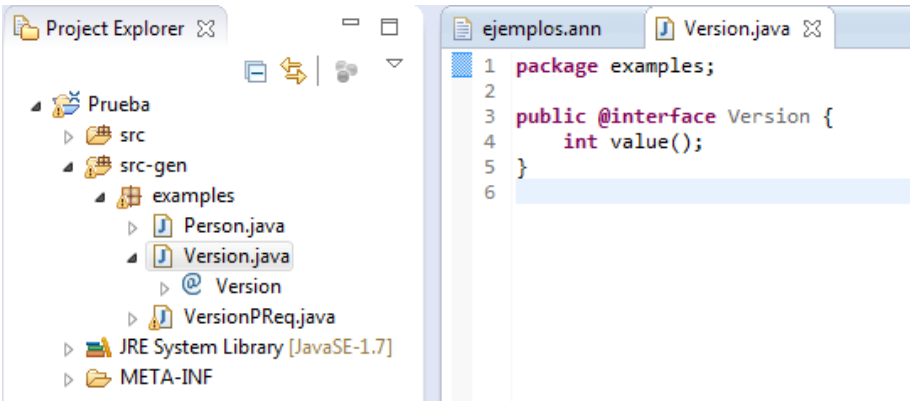

Figura 5.5. Código Java generado tras la definición de `Version`.

```
@Override
public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment objects) {

    for (Element e : objects.getElementsAnnotatedWith(Version.class)) {

        valid = false;
        m = e.getModifiers();
        message = "public class required.";
        if (e.getKind() == ElementKind.CLASS)
        {
            if (m.contains(Modifier.PUBLIC))
            {
                valid = true;
            }
        }

        if (valid == false)
        {
            this.processingEnv.getMessager().printMessage(Kind.ERROR,
                    "The annotation @Version is disallowed for this location. " + message, e);
        }
    }

    return false;
}
```

Figura 5.6. Método `process` del procesador de `Require` de `Person`.

La implementación del validador consiste en una serie de *Checks* que se realizan sobre los elementos; por ejemplo en la Figura 5.7 podemos ver un fragmento de un *Check* sobre los *statements* de una restricción. En concreto la parte que se muestra se encarga de comprobar que una clase abstracta no sea descrita como `final` en un *statement*, ya que esto en Java no puede suceder.

Podemos observar que tras detectar dicha condición se notifica el error mediante la función `error`, siendo uno de los parámetros proporcionados `F_MODIFIER`. Esto hace referencia al *quickfix* para este error. Un *quickfix* es, como su propio nombre indica, un método rápido para solucionar un error. Se proporciona al usuario una lista de estas posibles soluciones (en caso de existir más

```
37    /**
38     * Comprueba todas las restricciones asociadas a un statement.
39     */
40    @Check
41    def checkConstraintsStatement(StatementImpl st)
42    {
43        // Una clase no puede ser final y abstract al mismo tiempo
44        if (st.t_type == TargetType.CLASS && st.eIsSet(ConstraintPackage.STATEMENT__TMODS))
45        {
46            if (st.t_mods.abstract && st.t_mods.final)
47            {
48                error("An abstract class cannot be final",
49                    st.t_mods,
50                    ConstraintPackage$Literals::MODIFIERS__FINAL,
51                    F_MODIFIER,
52                    "final")
53            }
54        }
```

Figura 5.7. Fragmento de *Check* para *statement*.

de una) o *quickfixes* para que elija.

La implementación de todos los *Checks* es análoga: se recorre el elemento del modelo a validar en busca de los campos pertinentes y en caso de encontrar incoherencias se notifica al usuario. Además, en los casos que corresponda se proporcionan uno o varios *quickfixes* para solventar los errores. Entre las comprobaciones que realiza el validador encontramos:

- **Integridad de restricciones**. Incluye, por ejemplo, que no se prohíba algo que por otro lado se esté requiriendo. Se notifica al usuario y, en este caso, se proporciona el *quickfix* que elimina una de las restricciones que entran en contradicción.
- **Redundancia de restricciones**. Puede haber restricciones que contengan a otras; por ejemplo, `require public class` contiene a `require class`. En este caso también se notifica y se proporciona el *quickfix* que elimina la restricción innecesaria o redundante.
- **Coherencia de *statements***. Como el caso del ejemplo que hemos visto de una clase abstracta.

Veamos un uso de *quickfix* utilizando el ejemplo propuesto al inicio de la sección. En las Figuras 5.8, 5.9 y 5.10 vemos cómo se notifica el fallo, se propone un *quickfix* y se ejecuta, eliminando el error.

Por último, veamos un ejemplo de detección de incoherencia entre un `Require` y un `Forbid` en la Figura 5.11. En ella aparece una anotación para la que se ha prohibido anotar a clases, pero posteriormente se requiere que anote únicamente clases abstractas.

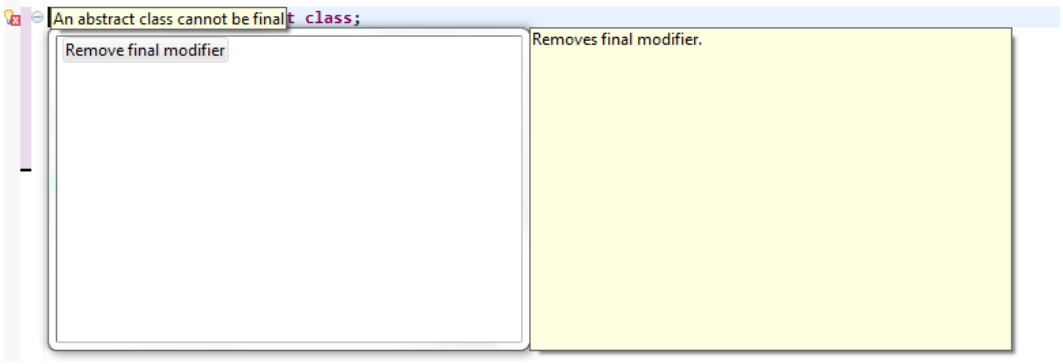

Figura 5.8. Notificación de *quickfix*.

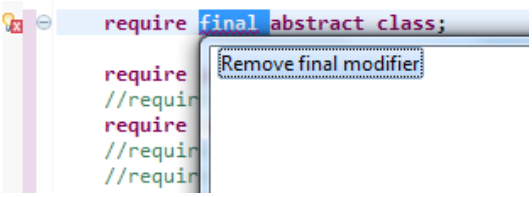

Figura 5.9. Selección de *quickfix*.

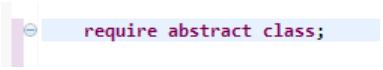

Figura 5.10. Solución gracias al *quickfix*.

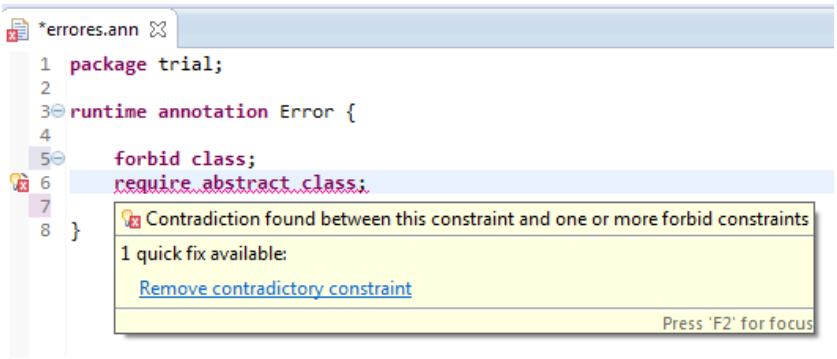

Figura 5.11. Error de incoherencia de restricciones.

6. Modelado de un ejemplo real: JPA

Finalmente, estudiaremos un caso real: un conjunto de anotaciones de JPA que describiremos mediante **Ann**. La información para este capítulo se ha extraído de la documentación oficial de JPA [15] [16] [17].

JPA es una API que permite la creación de esquemas relacionales para la persistencia de objetos Java. Las anotaciones seleccionadas de dicha API para las pruebas han sido `Entity`, `Id`, `IdClass`, `Embeddable` y `EmbeddedId`. Se han seleccionado por ser de frecuente uso en el contexto JPA ya que entre todas sirven para describir entidades y sus claves primarias, conceptos centrales en el diseño de bases de datos. Además, presentan un conjunto rico de restricciones entre ellas.

6.1. Características del conjunto de anotaciones

La anotación `Entity` representa una entidad, por lo que sólo puede aparecer anotando clases. Además, los requisitos estructurales para las clases anotadas con `Entity` son:

1. Debe tener un constructor `public` o `protected` sin argumentos. Puede tener otros constructores.
2. No puede ser `final`.
3. Ningún método puede ser `final`.
4. Las variables de instancia persistentes deben ser declaradas `private`, `protected` o con visibilidad de paquete.
5. Debe tener una clave primaria.

El último requisito es importante expandirlo, dado que es en el que entran en juego el resto de anotaciones propuestas.

La anotación `Id` sirve para señalar una clave primaria de una entidad. Únicamente puede ser aplicada por tanto en campos y métodos de clases anotadas con `Entity`.

Para representar una clave primaria compuesta se puede utilizar una clase que contenga los campos que forman la clave primaria. Esta clase puede ser anotada con `Embeddable`, en cuyo caso representa una clase cuyas instancias son componentes intrínsecos de la entidad original, y comparten clave primaria con la misma. También se puede utilizar una clase que simplemente represente los campos de la clave primaria, pero cumpliendo algunos otros requisitos.

En el primer caso, la clase `Embeddable` se utiliza como campo en la entidad y se anota con `EmbeddedId` para señalizarla como clave primaria. En la Figura 6.1 podemos encontrar un ejemplo de esto.

```
@Embeddable
public class EmployeePK implements Serializable {
    private String name;
    private long id;

    public EmployeePK() {
    }
    /** Getters and Setters **/
    ...
}

@Entity
public class Employee implements Serializable {
    @EmbeddedId EmployeePK primaryKey;

    public Employee() {
    }
    /** Getters and Setters **/
    ...
}
```

Figura 6.1. Clave primaria con `EmbeddedId`.

En el segundo caso, al definir la entidad se utiliza la anotación `IdClass` para indicar la clase que constituye la clave primaria; y cada uno de los campos de la misma se añaden a la entidad utilizando la anotación `Id`. En la Figura 6.2 hay un ejemplo de uso de esta alternativa (utilizando la misma clase para la clave compuesta que en el caso anterior pero sin anotar).

```
@IdClass(EmployeePK.class)
@Entity
public class Employee {
    @Id String empName;
    @Id long id;
...
}
```

Figura 6.2. Clave primaria con `IdClass`.

Es importante notar que una clase no puede tener a la vez un campo o método marcado con `Id` y otro con `EmbeddedId`, ya que esto implicaría dos claves primarias.

6.2. Definición mediante el DSL

Una vez realizada la descripcion del conjunto de anotaciones seleccionado, pasaremos a transcribirlo en el lenguaje **Ann**.

En la Figura 6.3 podemos ver la anotación `Entity`. Una mejora que observamos a simple vista respecto de Java es que podemos indicar que solamente anota clases, cuando en Java mediante el uso de `ElementType.TYPE` (ver Sección 3.1.1) estamos incluyendo también las interfaces y los enumerados.

Las restricciones que aparecen son autoexplicativas, y se corresponden con la mayoría de las que hemos mencionado en el apartado anterior. De entre ellas, hay alguna que no se ha conseguido representar mediante **Ann**:

- De la primera restricción sobre una clase anotada con `Entity` no se ha conseguido imponer que el constructor sea sin argumentos. Esto es así porque dicha característica no entra dentro del nivel de descripción de elementos Java al que actualmente se encuentra **Ann**.

- No se ha podido expresar la restricción 4 para una clase anotada con `Entity`. Para entender por qué primero es necesario analizar el requisito. Una variable de instancia persistente es cualquier campo dentro de una clase anotada con `Entity` que no sea ni `static` ni `final` ni esté marcado con `Transient`. Por tanto con la sintaxis actual de **Ann** no es posible expresar un requisito condicional sobre elementos de esta forma.

A pesar de estas dos carencias, observamos que ha sido posible expresar muchas restricciones que de otro modo quedarían relegadas a la documentación, como ocurre actualmente.

```
runtime annotation Entity {
    String name = "";

    require class;
    forbid final class;

    at class: require public constructor or protected constructor;
    at class: forbid final method;
    at class: require @Id method or @Id field or @EmbeddedId method or @EmbeddedId field;
    at class: forbid @Id method and @EmbeddedId method;
    at class: forbid @Id field and @EmbeddedId field;
}
```

Figura 6.3. Anotación `Entity` en **Ann**.

Por otro lado, las anotaciones `Embeddable` y `EmbeddedId` aparecen transcritas en la Figura 6.4. Vemos en la primera expresado el hecho de que al ser una clase que comparte la clave primaria

de la entidad en la que se encuentra embebida, no puede tener ella misma clave primaria. Además, para la anotación `EmbeddedId` se requiere también que se encuentre dentro de una clase anotada con `Entity`, dado que es donde tiene sentido únicamente.

```
runtime annotation Embeddable {
    require class;

    at class: forbid @Id method;
    at class: forbid @Id field;
}


runtime annotation EmbeddedId {
    require method or field;

    at field: require @Entity class;
    at method: require @Entity class;
}
```

Figura 6.4. Anotaciones `Embeddable` y `EmbeddedId` en **Ann**.

Por último, en la Figura 6.5 vemos el código **Ann** correspondiente a las anotaciones `Id` e `IdClass`. Cabe destacar que en la segunda aparece el requisito de que anota únicamente entidades, i. e., clases anotadas con `Entity`.

```
runtime annotation Id {
    require method or field;

    at field: require @Entity class;
    at method: require @Entity class;
}

runtime annotation IdClass {
    Class value;

    require @Entity class;
}
```

Figura 6.5. Anotaciones `Id` e `IdClass` en **Ann**.

Tras la definición de todas estas anotaciones, se ha generado el código Java correspondiente, como vemos en la Figura 6.6.

Vemos como salvo algunas particularidades se ha abarcado un conjunto muy amplio de

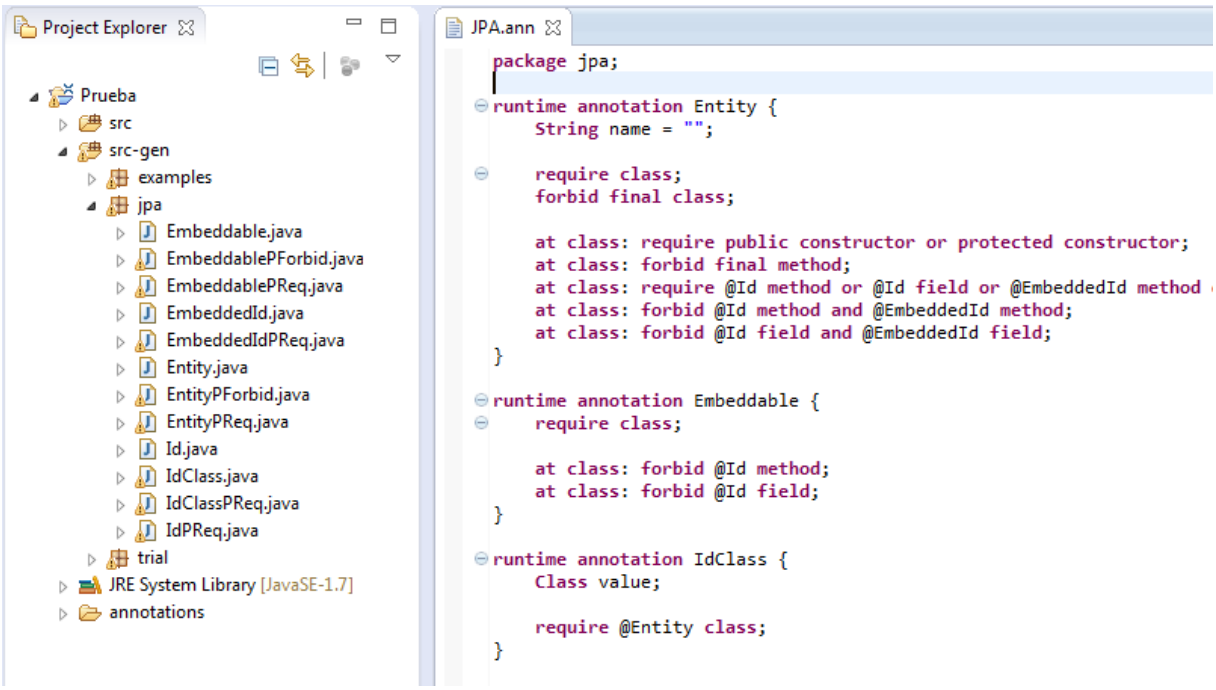

Figura 6.6. Código generado para las anotaciones JPA.

restricciones entre las anotaciones seleccionadas, representando satisfactoriamente el modelo subyacente.

6.3. Utilización

Una vez definidas las anotaciones y generado el código Java asociado, vamos a utilizar dicho código en un proyecto que utilice las anotaciones descritas para verificar su correcto funcionamiento.

Para exportar las anotaciones y sus procesadores y después poder utilizarlos en un proyecto Java ya existente, primero es necesario un paso previo: registrar los procesadores. Para ello en la carpeta *META-INF* del proyecto **Ann** hay que añadir el fichero que se muestra en la Figura 6.7.

Ahora ya estamos listos para exportar el código generado como un fichero `JAR`. Una vez exportado (en la Figura 6.7 también se puede ver dicho fichero), únicamente tenemos que incluirlo como librería en el proyecto en el que queramos usarlo. Es importante también activar el procesamiento de anotaciones y añadirlo en las opciones, tal y como se muestra en la Figura 6.8.

Finalmente, ya sólo queda utilizar las anotaciones importadas. En la Figura 6.9 vemos un ejemplo de una entidad llamada Persona. Observamos gracias al `import` que efectivamente está utilizando las anotaciones importadas. Además, no produce ningún error por cumplir todos los

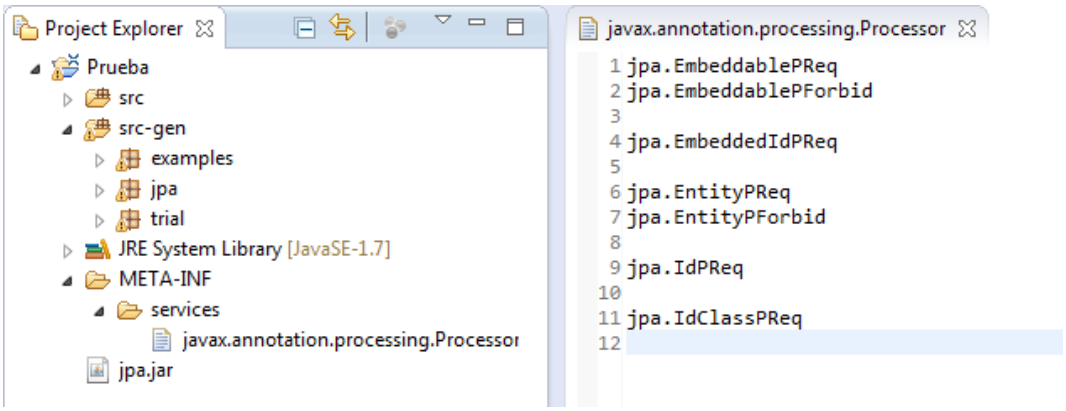

Figura 6.7. Registro de los procesadores generados.

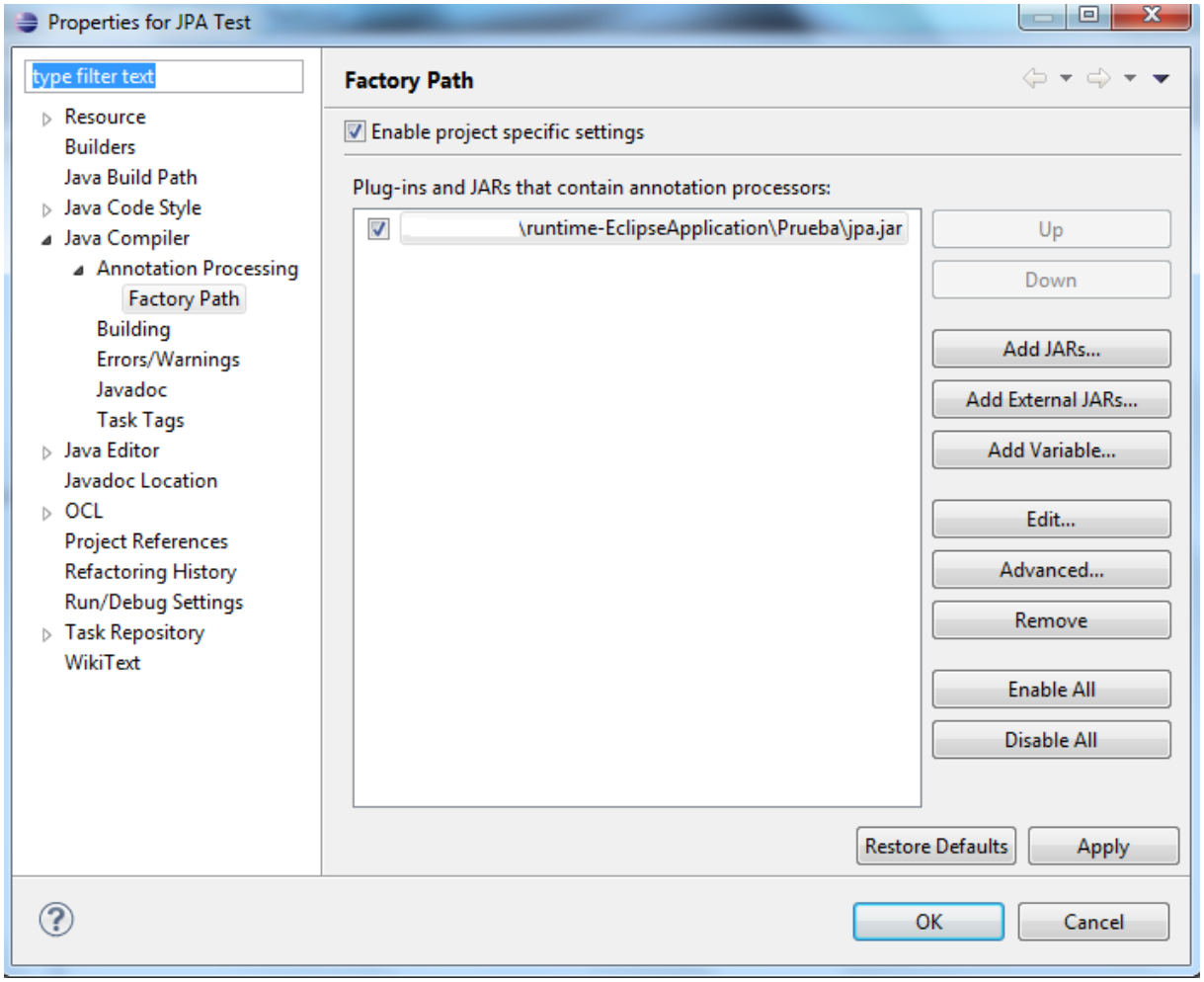

Figura 6.8. Activación del procesamiento de anotaciones en el proyecto.

requisitos vistos en los apartados anteriores.

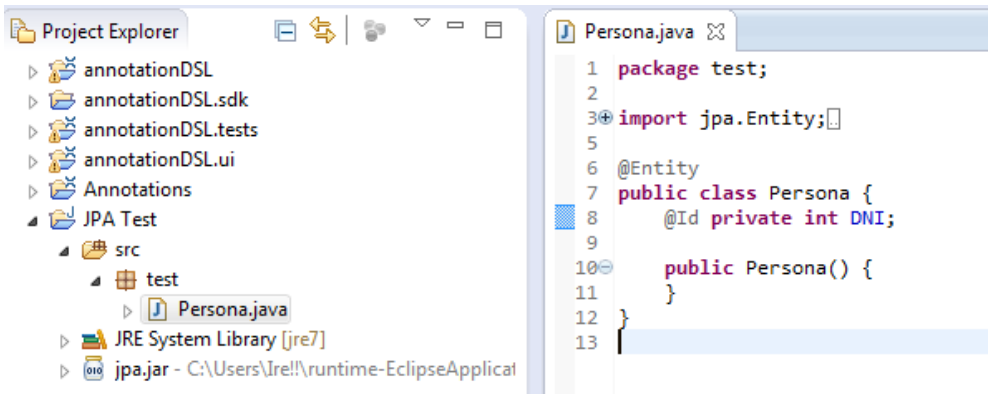

Figura 6.9. Uso correcto de las anotaciones en la entidad Persona.

Sin embargo, si ahora eliminamos la anotación `Id` o eliminamos la anotación `Entity`, nos aparece un error al no cumplirse ya el modelo descrito, como vemos en las Figuras 6.10 y 6.11.

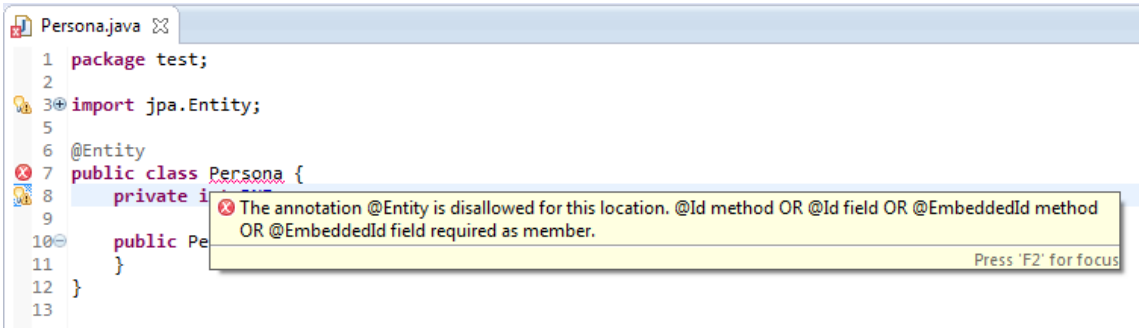

Figura 6.10. Error al eliminar la clave primaria.

Figura 6.11. Error al utilizar `Id` dentro de una clase no entidad.

Y si queremos utilizar por ejemplo `IdClass` y la clase no es una entidad, también obtenemos un error, como aparece en la Figura 6.12 (junto con el error en el `Id` anterior, ya que no está dentro de una entidad de nuevo).

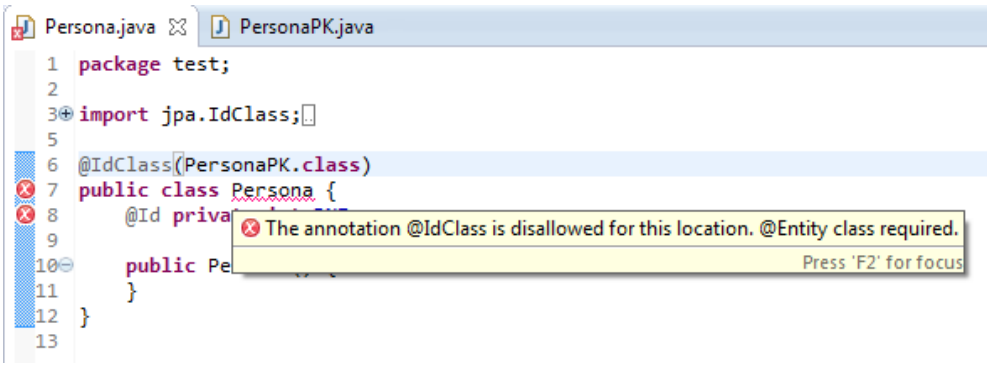

Figura 6.12. Error al utilizar `IdClass` en una clase no entidad.

De este modo ocurre cada vez que se viola una restricción.

7. Conclusiones

Ann permite el diseño efectivo de anotaciones Java, mejorando el soporte sintáctico básico que el lenguaje ofrece y permitiendo expresar diversas restricciones de integridad entre distintas anotaciones de un conjunto, como hemos podido comprobar en el Capítulo 6.

Además, gracias a la generación de código se permite la completa integración con proyectos Java ya existentes.

A la vista de todo esto, en este capítulo analizaremos el avance de **Ann** respecto a los otros enfoques vistos en el Capítulo 3, y posteriormente plantearemos las lineas de trabajo futuro.

7.1. Comparación con otras herramientas

En el Capítulo 3 vimos en detalle otras herramientas con objetivos similares a **Ann**: AnnaBot y AVal.

Ann proporciona además de un mecanismo de expresión de restricciones como AnnaBot, una nueva sintaxis para la definición de anotaciones más similar a la habitual en Java para la declaración de elementos contenedores. En este contexto, AnnaBot no hace ningún avance, centrándose más en la introducción de restricciones para conjuntos de anotaciones ya existentes. Además, aunque AnnaBot proporcione un DSL para la declaración de restricciones, como vimos actualmente únicamente es posible escribirlas en Java por no estar implementado el generador de código.

Actualmente en **Ann** la restricciones se describen dentro de la definición de la anotación a la que afectan, pero podrían ser fácilmente separables de dicha definición, simplemente indicando la anotación a la que se refieren con una cláusula adicional. Esto permitiría poder definir restricciones sobre conjuntos de anotaciones ya existentes, al igual que ocurre con AnnaBot, y utilizando el mismo soporte ya ofrecido.

Por otro lado, en el Capítulo 3 vimos que AnnaBot era capaz de expresar restricciones del tipo *no se puede anotar un elemento con ninguna anotación de este paquete*. Esto a día de hoy no es posible de expresar con **Ann**, si bien en un futuro se podría introducir mediante algún tipo de sintaxis con nombres completos de dominio y caracteres comodines a la hora de describir un *statement*.

Por último, **Ann** avisa en tiempo de compilación de las violaciones de restricciones que encuentra en los elementos de la aplicación Java, mientras que AnnaBot recurre a la reflexión, que ya vimos en el Capítulo 3 que era mejor evitar siempre que fuese posible por diversos motivos.

En cuanto a AVal, dado que utiliza meta-anotaciones para la descripción de restricciones, las posibilidades expresivas quedan limitadas respecto a la utilización de un DSL.

Además, **Ann** utiliza un conjunto pequeño de elementos, restricciones `Require` y `Forbid` con sintaxis uniforme pero semántica variada; mientras que AVal necesita de una cantidad mayor de meta-anotaciones para suplir esa diferencia expresiva. De hecho, aun esa mayor cantidad de elementos no es suficiente, ya que por ejemplo **Ann** es capaz de expresar condiciones acerca de los elementos de un contenedor anotado con cierta anotación, mientras que AVal es incapaz.

Por último, AVal se apoya en los procesadores Spoon mientras que **Ann** utiliza el soporte nativo de Java.

7.2. Trabajo futuro

En la sección anterior ya hemos podido entrever alguna de las posibles líneas de trabajo futuro.

Dado que **Ann** es un *framework* extensible al disponer de un sistema de restricciones uniforme en el que se pueden añadir subtipos a medida que se necesiten (actualmente únicamente `Require` y `Forbid`), si con la utilización se hiciera evidente que algún tipo de restricción frecuente no está siendo expresable, se podría añadir fácilmente al lenguaje. Hasta ahora, como hemos visto en el Capítulo 6, se dispone de un soporte muy completo, si bien no totalmente exhaustivo, pero que cubre la mayoría de restricciones importantes que existen en un conjunto de anotaciones.

Por otro lado, a lo largo de todo el documento se ha hecho énfasis en que se ha tomado un meta-modelo simplificado pero que cubría los casos más comunes para mostrar las posibilidades expresivas que subyacen al diseño de anotaciones. En este contexto una posible línea de trabajo futuro consistiría en añadir estos detalles menos frecuentes a cada una de las partes donde se han omitido, para que emule de forma totalmente fiel a todas las características y matices que presenta el lenguaje Java.

Además, el DSL siempre se podría extender para contemplar características adicionales del paradigma de orientación a objetos (recordamos la anotación `composite` del Capítulo 3), como pueden ser la herencia, el polimorfismo, la composición, etc. En definitiva, completar el DSL para que constituyese una integración completa de las anotaciones con el resto de contenedores del lenguaje Java.

Por último, y ya no encuadrado dentro de las extensiones posibles al meta-modelo de **Ann**, se podría dar soporte a la ingeniería inversa de anotaciones, generando un modelo a partir de una implementación Java de anotaciones.

A. Sintaxis concreta textual completa

En este anexo se incluye la sintaxis textual completa de **Ann** en EBNF.

Atributos

⟨*Attribute*⟩ ::= ⟨*ClassAtt*⟩
| ⟨*StringAtt*⟩
| ⟨*ExternalAtt*⟩
| ⟨*IntAtt*⟩
| ⟨*LongAtt*⟩
| ⟨*ShortAtt*⟩
| ⟨*FloatAtt*⟩
| ⟨*DoubleAtt*⟩
| ⟨*ByteAtt*⟩
| ⟨*CharAtt*⟩
| ⟨*BooleanAtt*⟩;

⟨*ClassAtt*⟩ ::= '`Class`' ('`[]`')? ⟨*ID*⟩ ('`=`' ⟨*ClassDefault*⟩)?;

⟨*StringAtt*⟩ ::= '`String`' ('`[]`')? ⟨*ID*⟩ ('`=`' ⟨*STRING*⟩)?;

⟨*ExternalAtt*⟩ ::= ⟨*ID*⟩ ('`[]`')? ⟨*ID*⟩ ('`=`' (⟨*EnumDefault*⟩ | ⟨*AnnDefault*⟩))?;

⟨*IntAtt*⟩ ::= '`int`' ('`[]`')? ⟨*ID*⟩ ('`=`' ⟨*INT*⟩)?;

⟨*LongAtt*⟩ ::= '`long`' ('`[]`')? ⟨*ID*⟩ ('`=`' ⟨*INT*⟩)?;

⟨*ShortAtt*⟩ ::= '`short`' ('`[]`')? ⟨*ID*⟩ ('`=`' ⟨*INT*⟩)?;

⟨*FloatAtt*⟩ ::= '`float`' ('`[]`')? ⟨*ID*⟩ ('`=`' ⟨*FLOAT*⟩)?;

⟨*DoubleAtt*⟩ ::= '`double`' ('`[]`')? ⟨*ID*⟩ ('`=`' ⟨*FLOAT*⟩)?;

⟨*CharAtt*⟩ ::= '`char`' ('`[]`')? ⟨*ID*⟩ ('`=`' ⟨*CHAR*⟩)?;

⟨*BooleanAtt*⟩ ::= '`boolean`' ('`[]`')? ⟨*ID*⟩ ('`=`' ⟨*BOOLEAN*⟩)?;

⟨*ByteAtt*⟩ ::= '`byte`' ('`[]`')? ⟨*ID*⟩ ('`=`' ⟨*BYTE*⟩)?;

⟨*AnnDefault*⟩ ::= ⟨*AnnID*⟩
| ⟨*AnnID*⟩ '`(`' ⟨*AnnValue*⟩ '`)`'
| ⟨*AnnID*⟩ '`(`' ⟨*KeyValue*⟩ ('`,`' ⟨*KeyValue*⟩)* '`)`';

⟨*ClassDefault*⟩ ::= ⟨*ID*⟩ '`.class`';

⟨*EnumDefault*⟩ ::= ⟨*ID*⟩ '`.`' ⟨*ID*⟩;

⟨*AnnID*⟩ ::= '`@`' ⟨*ID*⟩;

⟨*KeyValue*⟩ ::= ⟨*ID*⟩ '`=`' ⟨*AnnValue*⟩;

⟨*AnnValue*⟩ ::= ⟨*AnnArray*⟩
| ⟨*AnnBasicValue*⟩;

⟨*AnnArray*⟩ ::= '`{`' '`}`'
| '`{`' ⟨*AnnBasicValue*⟩ ('`,`' ⟨*AnnBasicValue*⟩)* '`}`';

⟨*AnnBasicValue*⟩ ::= ⟨*EnumDefault*⟩
| ⟨*AnnDefault*⟩
| ⟨*FLOAT*⟩
| ⟨*INT*⟩
| ⟨*BOOLEAN*⟩
| ⟨*CHAR*⟩
| ⟨*BYTE*⟩
| ⟨*STRING*⟩;

Anotaciones

⟨*Annotation*⟩ ::= ⟨*Retention*⟩? '`annotation`' ⟨*ID*⟩ '`{`' (⟨*Attribute*⟩ '`;`')* ⟨*Constraints*⟩?
'`}`';

⟨*Constraints*⟩ ::= (⟨*Require*⟩ | ⟨*Forbid*⟩)+;

⟨*Retention*⟩ ::= '`runtime`'
| '`class`'
| '`source`';

Restricciones

⟨*Forbid*⟩ ::= ‘`forbid`’ ⟨*Statement*⟩ (‘`and`’ ⟨*Statement*⟩)* ‘`;`’
| ‘`at`’ ⟨*TargetType*⟩ ‘`:`’ ‘`forbid`’ ⟨*Statement*⟩ (‘`and`’ ⟨*Statement*⟩)* ‘`;`’;

⟨*Require*⟩ ::= ‘`require`’ ⟨*Statement*⟩ (‘`or`’ ⟨*Statement*⟩)* ‘`;`’
| ‘`at`’ ⟨*TargetType*⟩ ‘`:`’ ‘`require`’ ‘`all`’? ⟨*Statement*⟩ (‘`or`’ ⟨*Statement*⟩)* ‘`;`’;

⟨*Statement*⟩ ::= ⟨*AnnID*⟩
| ⟨*TgtStatement*⟩;

⟨*TgtStatement*⟩ ::= ⟨*AnnID*⟩? ⟨*Modifiers*⟩ ⟨*TargetType*⟩;

⟨*Modifiers*⟩ ::= ⟨*VisibMod*⟩? & ‘`final`’? & ‘`abstract`’? & ‘`static`’?;

⟨*VisibMod*⟩ ::= ‘`public`’
| ‘`private`’
| ‘`protected`’
| ‘`package`’;

⟨*TargetType*⟩ ::= ‘`interface`’
| ‘`class`’
| ‘`annotation`’
| ‘`method`’
| ‘`field`’
| ‘`constructor`’
| ‘`enum`’ ;

B. Fragmentos de implementación

A continuación se muestran fragmentos representativos de la implementación de los distintos componentes de **Ann**, tal y como se ha explicado en el Capítulo 5. Aunque algunas implementaciones aparecen completas, no es lo habitual.

En el apartado correspondiente a los meta-modelos se han omitido tipos básicos como enumerados o elementos muy similares y análogos a los ya mostrados, que se pueden inferir a partir del meta-modelo y los elementos presentes.

Para la implementación de la sintaxis concreta textual se han omitido reglas básicas como las correspondientes a valores de enumerados y tipos de datos, y reglas análogas a las ya mostradas que se pueden inferir a partir de la gramática y las reglas presentes.

Meta-modelo en Emfatic

Anotaciones

```
class Annotation {
  attr String name;
  val attribute.Attribute[*] attributes;
  attr Retention retention;
  val Constraints[?] constraints;
}

class Constraints {
    val constraint.Require[*] require;
    val constraint.Forbid[*] forbid;
}

enum Retention {
  not_set = 0;
  runtime = 1;
  ~class = 2;
  source = 3;
}
```

Atributos

```
abstract class Attribute {
  attr String name;
  attr boolean is_array;
}

class ClassAtt extends Attribute {
    attr String [?] default;
}

class StringAtt extends Attribute {
    attr String [?] default;
}

class ExternalAtt extends Attribute {
  attr String type;
  attr String[?] default;
}

abstract class PrimitiveAtt extends Attribute {
}
```

Restricciones

```
abstract class Constraint {
    val Statement[*] stmts;
    attr TargetType[?] t_type;
}

class Require extends Constraint {
    attr boolean all;
}

class Forbid extends Constraint {
}

class Statement {
    ref annotation.Annotation[?] t_ann;
    val Modifiers[?] t_mods;
    attr TargetType[?] t_type;
}
```

Sintaxis textual en Xtext

Anotaciones

```
Annotation returns Annotation:
    (retention=Retention)? "annotation" name=EString
    "{"
        (attributes+=Attribute ";")*
        (constraints=Constraints)?
    "}";

Constraints returns Constraints:
    (require+=Require | forbid+=Forbid)+
;
```

Atributos

```
Attribute returns Attribute:
    ClassAtt | StringAtt | ExternalAtt |
    IntAtt | LongAtt | ShortAtt | FloatAtt | DoubleAtt | ByteAtt | CharAtt | BooleanAtt;

ClassAtt returns ClassAtt:
    "Class" (is_array?="[]")? name=EString ("=" default=ClassDefault)?;

StringAtt returns StringAtt:
    "String" (is_array?="[]")? name=EString ("=" default=EString)?;

ExternalAtt returns ExternalAtt:
    type=EString (is_array?="[]")? name=EString ("=" (default=(AnnotationDefault | EnumDefault)))?;

IntAtt returns IntAtt:
    "int" (is_array?="[]")? name=EString ("=" default=EInt)?;

LongAtt returns LongAtt:
    "long" (is_array?="[]")? name=EString ("=" default=ELong)?;

ShortAtt returns ShortAtt:
    "short" (is_array?="[]")? name=EString ("=" default=EShort)?;

AnnotationDefault returns ecore::EString:
    AnnID |
    AnnID "(" AnnValue ")" |
    AnnID "(" KeyValue ("," KeyValue)* ")"
;
```

Restricciones

```
Forbid returns Forbid:
    "forbid" stmts+=Statement ("and" stmts+=Statement)* ";" |
    "at" t_type=ContainerType ":"
        "forbid" stmts+=InnerStatement ("and" stmts+=InnerStatement)* ";" |
    "at" t_type=InnerType ":"
        "forbid" stmts+=ContainerStatement ";"
;

Require returns Require:
    "require" stmts+=Statement ("or" stmts+=Statement)* ";" |
    "at" t_type=ContainerType ":"
        "require" (all?="all")? stmts+=InnerStatement ("or" stmts+=InnerStatement)* ";" |
    "at" t_type=InnerType ":"
        "require" stmts+=ContainerStatement ("or" stmts+=ContainerStatement)* ";"
;


Statement returns Statement:
    AnnStatement | TgtStatement
;

TgtStatement returns Statement:
    ClassStatement | InterfaceStatement | AnnotationStatement | EnumStatement |
    FieldStatement | MethodStatement | ConstructorStatement
;

InnerStatement returns Statement:
    FieldStatement | MethodStatement | ConstructorStatement
;

ContainerStatement returns Statement:
    ClassStatement | InterfaceStatement | AnnotationStatement
;

ContainerType returns TargetType:
    "interface" | "class" | "annotation"
;

InnerType returns TargetType:
    "method" | "constructor" | "field"
;
```

```
AnnStatement returns Statement:
    "@" t_ann=[Annotation|ID]
;

ClassStatement returns Statement:
    ("@" t_ann=[Annotation|ID])? t_mods=ClassModifiers t_type=ClassType
;

InterfaceStatement returns Statement:
    ("@" t_ann=[Annotation|ID])? t_mods=InterfaceModifiers t_type=InterfaceType
;
```

Generador de código

Generador principal

```
13 class AnnotationGenerator implements IGenerator {
14
15     @Inject
16     AnnotationDefGen annDefGen;
17
18     @Inject
19     AnnotationPReqGen annPReqGen;
20
21     @Inject
22     AnnotationPForbidGen annPForbidGen;
23
24     /**
25      * @brief Se delega en distintos procesadores con una funcion especifica.
26      */
27     override void doGenerate(Resource resource, IFileSystemAccess fsa) {
28
29         annDefGen.doGenerate(resource, fsa);
30         annPForbidGen.doGenerate(resource, fsa);
31         annPReqGen.doGenerate(resource, fsa);
32     }
33 }
34
35
```

Template para la definición de una anotación

```
45    def compile(Annotation a) '''
46        package «model.package»;
47
48        «IF a.retention != Retention.NOT_SET»
49        import java.lang.annotation.Retention;
50        import java.lang.annotation.RetentionPolicy;
51
52        @Retention(RetentionPolicy.«a.retention.name.toUpperCase»)
53        «ENDIF»
54        public @interface «a.name» {
55            «FOR f: a.attributes»
56            «f.compile»
57            «ENDFOR»
58        }
59    '''
60
```

Fragmento de *template* para el procesador de `Forbid`

```
66      @Override
67      public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment objects) {
68
69          for (Element e : objects.getElementsAnnotatedWith(«a.name».class)) {
70
71              «FOR f : a.constraints.forbid»
72              valid = true;
73              «IF f.t_type == TargetType.DEFAULT»
74              «f.compile_no_tgt»
75              «ELSE»
76              if (e.getKind() == ElementKind.«f.t_type.name.toUpperCase»)
77              {
78                  «IF f.t_type.isContainer»
79                  le = e.getEnclosedElements();
80                  «f.compile_tgt_container»
81                  «ELSEIF f.t_type.isInner»
82                  enclosing = e.getEnclosingElement();
83                  «f.compile_tgt_inner»
84                  «ENDIF»
85              }
86              «ENDIF»
87              if (valid == false)
88              {
89                  this.processingEnv.getMessager().printMessage(Kind.ERROR,
90                          "The annotation @«a.name» is disallowed for this location."  + message, e);
91              }
92              «ENDFOR»
93          }
94
95          return false;
96      }
97  }
98  '''
99
```

Template para la comprobación de *statements* en `Require`

```
154    /**
155     * Los dos tipos de statement que existen son:
156     *      @<Annotation>? <modifiers>? <target>
157     *      @<Annotation>
158     */
159    def compile (StatementImpl s, String element) '''
160    «IF s.eIsSet(ConstraintPackage.STATEMENT__TTYPE)»
161    if («element».getKind() == ElementKind.«s.t_type.name.toUpperCase»)
162    {
163        «IF s.eIsSet(ConstraintPackage.STATEMENT__TANN)»
164        if («element».getAnnotation(«s.t_ann.name».class) != null)
165        {
166            «IF s.hasModifiersConstraints»
167            if («s.t_mods.compile»)
168            {
169                valid = true;
170            }
171            «ELSE»
172            valid = true;
173            «ENDIF»
174        }
175        «ELSEIF s.hasModifiersConstraints»
176        if («s.t_mods.compile»)
177        {
178            valid = true;
179        }
180        «ELSE»
181        valid = true;
182        «ENDIF»
183    }
184    «ELSEIF s.eIsSet(ConstraintPackage.STATEMENT__TANN)»
185    if («element».getAnnotation(«s.t_ann.name».class) != null)
186    {
187        valid = true;
188    }
189    «ENDIF»
190    '''
```

Acrónimos

API *Application Programming Interface*.

DSL *Domain-Specific Modeling Language*.

EBNF *Extended Backus–Naur Form*.

EJB *Enterprise JavaBean*.

EMF *Eclipse Modeling* Framework.

GPL *General-Purpose Modeling Language*.

IDE *Integrated Development Environment*.

JAX-WS *Java API for XML Web Services*.

JPA *Java Persistence API*.

MDD *Model-Driven Development*.

ORM *Object Relational Mapping*.

UML *Unified Modeling Language*.

Glosario

Java Reflection API API que proporciona Java para la reflexión.

framework Estructura conceptual y tecnológica de soporte definido, normalmente con módulos de software concretos, que puede servir de base para la organización y desarrollo de software.

plug-in Componente software que añade características específicas a una aplicación software ya existente.

target Elemento de un programa Java que puede ser anotado por una anotación.

web service Servicio que utiliza un conjunto de protocolos y estándares para intercambiar datos entre aplicaciones.

anotación de marcado Anotación que no contiene ningún miembro. Únicamente es significativa su presencia o ausencia.

Entorno de Desarrollo Integrado Aplicación software que ofrece facilidades para la programación en el desarrollo de software. Suele constar de un editor de código fuente con resaltado de sintaxis, herramientas para la automatización de la compilación y un depurador. La mayoría de IDEs modernos ofrecen además utilidades para completar el código.

lenguaje de modelado Herramienta conceptual para describir la realidad de forma explícita, ya sea textual o gráficamente.

meta-anotación Anotación que se utiliza en la declaración de otras anotaciones.

meta-dato Datos que describen a otros datos.

meta-modelo Abstracción de un modelo que expresa propiedades sobre el propio modelo.

modelo Representación simplificada o parcial de la realidad, definida para llevar a cabo una tarea o llegar a un acuerdo sobre un tema.

procesador de anotaciones Extensiones al compilador de Java que procesan las anotaciones de un programa y pueden realizar diversas acciones en consecuencia, como generar errores y warnings en el compilador, escribir ficheros,....

reflexión Habilidad de un programa de consultar y modificar su estructura y comportamiento en tiempo de ejecución.

retención Cualidad de una anotación que indica su tiempo de visibilidad durante el desarrollo de un programa.

Bibliografía

[1] David Flanagan, ***Java in a Nutshell***. O'Reilly, Sebastopol, 5th Edition, 2005.

[2] Ian F. Darwin, ***Java Cookbook***. O'Reilly, Sebastopol, 2nd Edition, 2004.

[3] Joshua Bloch, ***Effective Java***. Addison-Wesley, Upper Saddle River, 2nd Edition, 2008.

[4] Bruce Eckel, ***Thinking in Java***. Prentice Hall, Upper Saddle River, 4th Edition, 2006.

[5] Herbert Schildt, ***Java: The complete reference, J2SE***. McGraw-Hill, New York, 5th Edition, 2005.

[6] Marco Brambilla, Jordi Cabot and Manuel Wimmer, ***Model-Driven Software Engineering in Practice***. Morgan & Claypool Publishers, Synthesis Lectures on Software Engineering, 2012.

[7] Federico Mancini, Dag Hovland and Khalid A. Mughal, *Investigating the limitations of Java annotations for input validation.* Availability, Reliability, and Security, pages 513 - 518, 2010.

[8] Carlos Noguera and Renaud Pawlak, *AVal: an Extensible Attribute-Oriented Programming Validator for Java.* J. Softw. Maint. Evol. 19, 4 (July 2007), pages 253-275, 2007.

[9] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble, *JavaCOP: Declarative pluggable types for java.* ACM Trans. Program. Lang. Syst. 32, 2, Article 4 (February 2010), 37 pages.

[10] Ian Darwin, *AnnaBot: A Static Verifier for Java Annotation Usage.* Advances in Software Engineering, Volume 2010, Article ID 540547 2010.

[11] Viera K. Proulx and Weston Jossey, *Unit test support for Java via reflection and annotations.* In Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (PPPJ '09). ACM, New York, NY, USA, 49-56.

[12] Glauber Ferreira, Emerson Loureiro, and Elthon Oliveira, *A Java code annotation approach for model checking software systems.* In Proceedings of the 2007 ACM symposium on Applied computing (SAC '07). ACM, New York, NY, USA, 1536-1537.

[13] Walter Cazzola and Edoardo Vacchi, *@Java: annotations in freedom*. In Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13). ACM, New York, NY, USA, 1688-1693.

[14] Andrew Phillips, *@composite: macro annotations for Java C*. In Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications (OOPSLA '09). ACM, New York, NY, USA, 767-768.

[15] http://docs.oracle.com/javaee/7/tutorial/doc/persistence-intro001.htm#BNBQA

[16] http://docs.oracle.com/javaee/7/api/

[17] http://docs.oracle.com/cd/E16439_01/doc.1013/e13981/cmp30cfg001.htm

[18] http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html

[19] http://docs.oracle.com/javase/specs/jls/se7/html/jls-9.html#jls-9.6

[20] http://docs.oracle.com/javase/7/docs/api/javax/annotation/processing/Processor.html

[21] http://docs.oracle.com/javase/tutorial/reflect/

[22] http://docs.oracle.com/javase/tutorial/java/annotations/repeating.html

[23] http://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html

[24] http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html#details

[25] http://www.eclipse.org/xtend/

[26] http://docs.jboss.org/seam/2.3.1.Final/reference/html_single/#annotations

[27] http://docs.spring.io/spring/docs/3.0.0.M3/reference/html/ch29.html

[28] http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/

[29] http://spoon.gforge.inria.fr/