La construcción de un **modelo de dominio** es un proceso iterativo

**Brainstorming y refinamiendo**

**Un Modelo de Dominio borrador**

**Un Diagrama de Clases temprano**

**Feedback del Prototipo**

Este proceso se repetirá hasta que el modelo, el diagrama y el prototipo sean correctos.

La capa de negocio es la más importante porque es la única que no depende de la tecnología. Comúnmente se le llama **Aplicación** a lo que comprende las capas de presentación, acceso al negocio y persistencia, y**Negocio** específicamente a la capa de dominio.

**Tiers de Aplicación:**

* Presentación
* Acceso al Negocio
* Persistencia

**Tiers de Negocio:**

* Capa de dominio

**User Interface**

**(Presentation**

**Layer)**

**Application**

**Layer**

**Domain Layer**

**Infrastructure**

**Layer**

**El dominio no depende de la aplicación**

Este modelo o dominio, que no es más que un conjunto de objetos y clases interrelacionados al mejor estilo de la programación orientada a objetos, es el **negocio**, y **puede ser reutilizado en distintas aplicaciones**, puede ser desenganchado de una aplicación e injertado en otra, con una tecnología completamente diferente, como si se tratara de una pieza de Lego.

[**Domain-Driven Design**](http://en.wikipedia.org/wiki/Domain-driven_design)**(DDD) es un conjunto de buenas prácticas, patrones y recomendaciones para construir software sólido y robusto a partir de la creación del dominio.**

## Comunicación y el Uso del Lenguaje

El **lenguaje ubicuo** de un proyecto debe incluir:

* Nombres de clases y sus operaciones destacadas (sobre todo las relacionadas estrechamente con el negocio)
* Términos para discutir **reglas de negocio** que se han hecho explícitas en el modelo
* Nombres de **patrones de análisis y de diseño** que se aplican al modelo de dominio

El lenguaje ubicuo debe ser utilizado por todas las personas involucradas en el proyecto, inclusive programadores y expertos de dominio.

**El modelo de dominio es la columna vertebral del lenguaje ubicuo.**

Si los programadores descubren, por ejemplo, que una clase de dominio debe llamarse de otra forma, no pueden *refactorear* el nombre de la clase sin *refactorear* también el nombre del concepto en el modelo de dominio y en el lenguaje ubicuo.

**Patrones de dominio**

**I. Patrón 1: Layered Architecture**

**II. Anti-Patrón: Smart UI**

**III. Associations**

Para que el modelo sea más sencillo y fácil de implementar,**es importante limitar las asociaciones lo más posible.**Para esto, existen al menos tres criterios para simplificar una asociación:

1) Imponer una d

irección a la asociación

2) Añadir un cualificativo en algún extremo de la asociación (para reducir la multiplicidad)

3) Eliminar las asociaciones no esenciales

**IV. Patrón 2: Entities**

**Toda Entidad debe contar con un método para comparar su identidad con otra instancia de su misma clase.**

**Una Entidad es algo que tiene continuidad a través de un ciclo de vida y es distinguible independientemente de los atributos que son importantes para la aplicación.**

Ejemplos obvios de **Entities** pueden ser: Persona, Cliente, Factura, Vuelo, País, y un largo etcétera.

DDD recomienda modelar las Entidades lo más pequeñas y sencillas posible, añadiendo sólo el comportamiento y los atributos necesarios. Cualquier comportamiento y/o atributo que pueda ser trasladado a otra **Entidad** o a otro **Value Object**, bienvenido sea.

**V. Patrón 3: Value Objects**

Ejemplos de **Value Objects** pueden ser: Color, Dirección, Rueda, Lápicera, Herramienta, etc.

**Los Value Objects son instanciados para representar elementos del diseño de los cuales importa qué son y no quién son.**

Suelen ser usados como atributos de **Entities** u otros **Value Objects**.

**VI. Patrón 4: Services**

**Los Servicios son actividades, funciones, operaciones, no cosas.** No conservan estado; **son *stateless***.

Un **Servicio** debe tener una responsabilidad bien definida, y esa responsabilidad y su interface deben estar definidas como parte del Modelo de Dominio.

Los parámetros y los resultados del **Servicio**deben ser **objetos de dominio**.

Es muy común construir una capa de **Servicios** para manipular la capa de dominio. Es una forma de desacoplar el dominio de la presentación. Muchos de los **Servicios** de dominio pueden vivir en esta capa.

**VII. Patrón 5: Modules**

En general, los lenguajes de programación proveen una sintaxis para crear**Módulos**, y los IDEs hacen su parte mostrándonos esos módulos agrupados de forma visual. Si estamos en el mundo de Java, por ejemplo, estos módulos serán los paquetes.

**Los Módulos, en una capa de dominio, sirven para separar el negocio en áreas o incumbencias significativas y también para aislar la capa entera del resto del código.**

**IX. Patrón 6: Aggregates**

Ejemplo típico: una **Persona** (un**Entity**)que tiene una **Dirección** (un**Value Object**). Ambos objetos están asociados mediante una composición. La **Dirección** pertenece a la **Persona** y no tiene razón de ser fuera de la misma. No se actualiza una **Dirección**, se actualiza una **Persona**, **se actualiza la dirección de unaPersona.**

**Una Agregación es un grupo de objetos asociados que deben tratarse como una unidad a la hora de manipular sus datos.** En el caso del ejemplo, **Persona** y **Dirección** forman parte de una misma**Agregación**.

**X. Patrón 7: Factories**

Los **Factories** sirven para encapsular la creación de objetos complejos. La creación de una**Entidad**, por ejemplo, podría ser tan compleja que involucre a muchas otras **Entidades** y **Value Objects**. Imaginemos que se trata de la **Entidad Raíz** de una **Agregación**. Es evidente que su creación implica la creación de muchos otros objetos y el cumplimiento de reglas de negocio y consistencia que deben hacerse cumplir. Para abstraer al cliente y al objeto en sí de este complejo mecanismo es que se provee un **Factory**que se encargue del trabajo sucio.

**Mediante los Factories podemos crear Agregaciones enteras al momento de instanciar la Entidad Raíz, haciendo cumplir todas sus invariantes.**

Las implementaciones de **Factory** más populares son las de [Erich Gama](http://en.wikipedia.org/wiki/Erich_Gamma): [*Factory Method*](http://es.wikipedia.org/wiki/Factory_Method_%28patr%C3%B3n_de_dise%C3%B1o%29), *[Abstract Factory](http://es.wikipedia.org/wiki/Abstract_factory)*,[*Builder*](http://es.wikipedia.org/wiki/Builder_%28patr%C3%B3n_de_dise%C3%B1o%29) y *[Prototype](http://es.wikipedia.org/wiki/Prototype_%28patr%C3%B3n_de_dise%C3%B1o%29)*.

**XI. Patrón 8: Repositories**

El patrón **Repository** es un simple framework conceptual que encapsula el proceso de reconstitución y lo diferencia del proceso de creación encapsulado por el patrón **Factory**. **Un Factory construye nuevos objetos; un Repository busca objetos viejos, ya creados.**

 Es la forma que el modelo de dominio tiene de reconstituir **Entidades** y **Value Objects** y almacenarlos para que puedan sobrevivir más allá de la ejecución del programa. **La interfaz del Repository vivirá en la capa de dominio; la implementación es posible que viva en otra capa (por ejemplo en la de Persistencia).**

 Como regla general, **es casi seguro que tengamos un Repository por cada Entidad Raíz de cada Agregación**.

## Conclusión

Resumiendo, la Parte 2 de *[Domain-Driven Design: Tackling Complexity in the Heart of Software](http://www.amazon.com/exec/obidos/ASIN/0321125215/domainlanguag-20)* (ISBN: 0-321-12521-5) habla de los componentes fundamentales de un *Model-Driven Design*. Comienza con un patrón,**Layered Architecture**, que nos dice que una forma apropiada de **aislar el dominio** del resto de la aplicación es a través de una correcta separación de capas. Hay muchas formas de dividir en capas una aplicación, una de las más comunes es la que ya venimos viendo en los últimos posts: **Presentación / Acceso al Dominio / Dominio / Persistencia / Base de Datos**. En la capa de **Dominio** es donde vivirá toda la lógica de negocio y los datos sobre los cuales esta lógica opera.

Una vez aislado el dominio, Evans nos cuenta que existen **Asociaciones** entre los objetos de dominio y que es muy importante limitar la cantidad de ellas. Para ello nos da un conjunto de tips bastante útil a la hora de modelarlas. En el dominio pueden existir tres tipos de objetos: **Entities**, **Value Objects** y **Services**. Los**Entities** son objetos de dominio que tienen identidad y que por lo tanto son distinguibles. Los **Value Objects**no tienen identidad, por ende no son distinguibles; simplemente son objetos que representan algún valor o estado. Los **Services** son objetos puramente funcionales, que publican servicios para que sean consumidos por otros objetos de dominio o por la capa superior, y que manipulan **Entities** y **Value Objects**. Cuando un modelo crece bastante como para ser manejado de forma atómica, Evans nos enseña a separarlo en **Módulos**utilizando algún mecanismo provisto por el lenguaje de programación utilizado (en el caso de Java, los**Módulos** serían **Paquetes**).

A continuación, Evans se enfoca en el ciclo de vida de los objetos de dominio. Dice que un objeto nace, a través de un **Factory** o de un código simple de creación, se manipula en memoria y luego se almacena en un**Repositorio** de objetos de su mismo tipo. Este objeto puede ser vuelto a pedir al **Repositorio** en cualquier momento para volver a ser modificado y volverlo a guardar, archivarlo o borrarlo. Gracias a los **Repositorios**, el dominio puede abstraerse de la infraestructura que reside en la capa inferior y que se encarga de la**Persistencia** de los objetos.

Pero no todos los objetos serán accedidos directamente a través de **Repositorios**. Habrá objetos que se manipularán indirectamente, a través de **Asociaciones** establecidas con el objeto que sí se trajo del**Repositorio**. Así se establecen las **Agregaciones**, que encapsulan un grupo de **Entidades** y **Value Objects**que serán creados, modificados y borrados como una sola unidad. Vistas desde afuera, las **Agregaciones**cuentan con una **Entidad Raíz**, que es la que generalmente será recuperada desde un **Repositorio**, y una**Frontera**, que encerrará un conjunto de objetos internos que son invisibles al exterior. (Una aclaración que no he mencionado antes es que no siempre la raíz de una **Agregación** tiene que ser una **Entidad**; puede haber excepciones; pero el caso más común es que sea una **Entidad**, cuya identidad única le brinda una identidad única a la **Agregación** entera.)