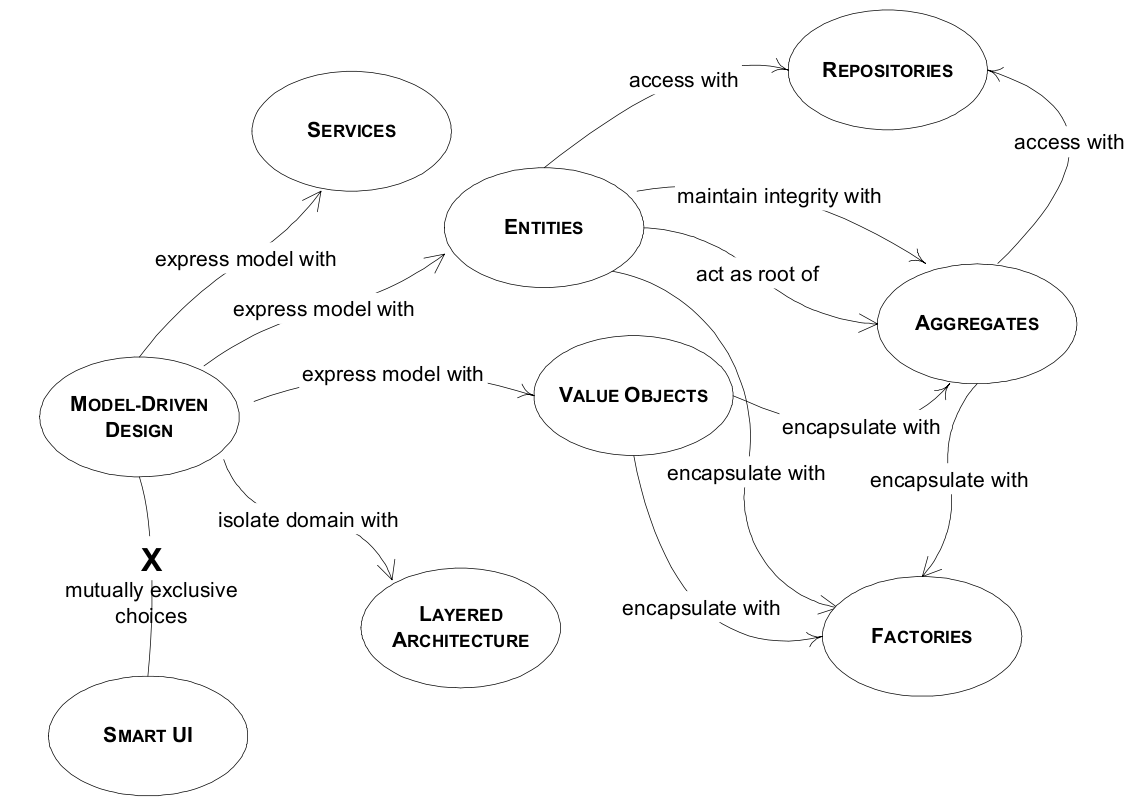
**MATERIAL DE ESTUDIO**

DDD: CONCEPTOS GENERALES

Desarrollar un buen Modelo de Dominio es un arte, pero la tarea de diseñar e implementar elementos individuales del modelo puede resultar relativamente sistemática.

5) **Eric Evans** propone seguir un conjunto de **patrones de dominio**, ya probados y estudiados, para conseguir modelar un dominio que resulte práctico para la implementación. En este post veremos estos patrones, que son parte del ecosistema de*Domain-Driven Design* y son los que Evans llama *"los componentes fundamentales de Model-Driven Design"*:

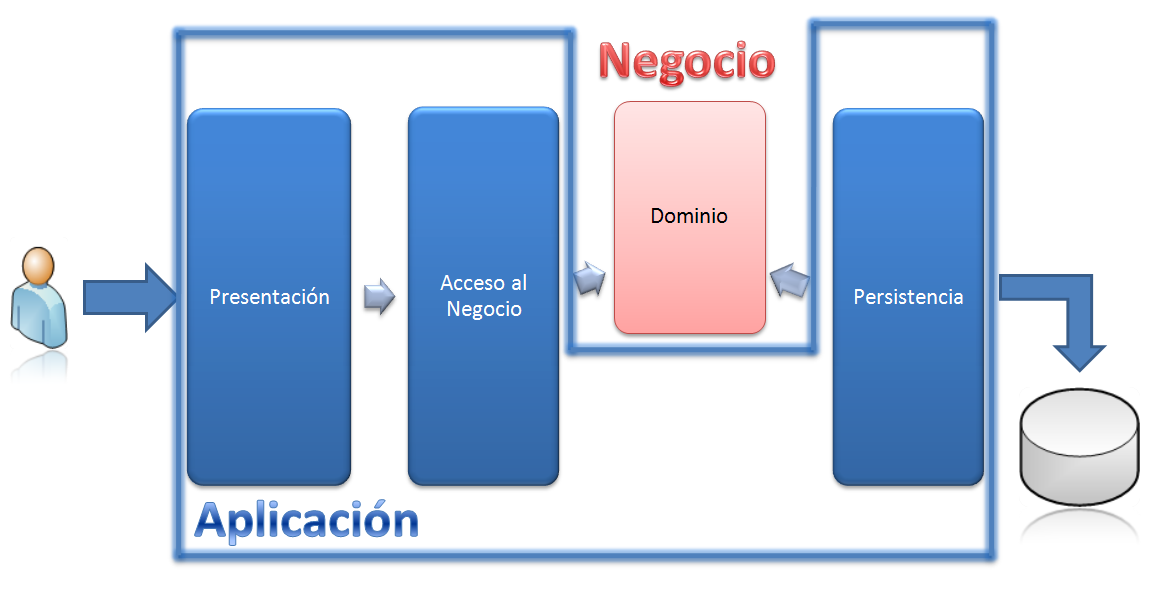


**Aislando el Dominio**

Usualmente, las líneas de código que efectivamente resuelven los problemas del dominio constituyen apenas una pequeña porción del sistema. Sin embargo, la importancia de estas líneas es absolutamente desproporcionada con respecto a su tamaño. Como vimos en un sistema grande **la importancia de separar los objetos de dominio del resto de la aplicación es extremadamente importante**. La tarea principal de un equipo de trabajo de DDD es aislar el dominio de los demás conceptos del sistema relacionados con la tecnología. **La estrategia de dividir las funcionalidades del sistema en capas es uno de los caminos más apropiados para desacoplar el dominio de la tecnología.**

1. **Patrón 1: Layered Architecture**

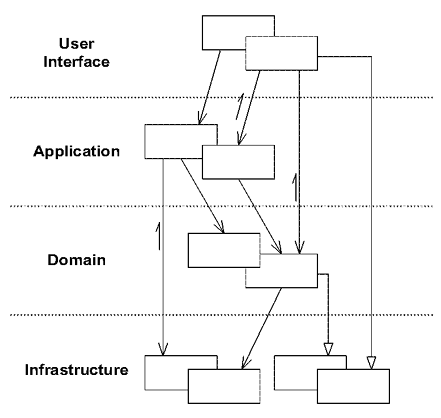
La separación de capas típica, que es la que propone **Martin Fowler** en su famoso libro *[Patterns of Enterprise Application Architecture](http://www.amazon.com/exec/obidos/ASIN/0321127420)*, la biblia fundamental para la construcción de Aplicaciones Enterprise y el nuevo testamento de los Patrones de Diseño:



Ésta no es la única forma de dividir en capas una aplicación existen varios otros esquemas de capas para arquitecturas enterprise:

* Presentación / Controlador / Dominio / Data Mapping / Data Source *[Capas de Brown]*
* Cliente / Presentación / Negocio / Integración / Recursos *[Capas del viejo core de J2EE]*
* Presentación / Negocio / Acceso a Datos *[Capas de Microsoft DNA]*
* Presentación / Aplicación / Servicios / Dominio / Persistencia *[Capas de Marinescu]*
* Consumer / Consumer Helper / Aplicación / Dominio / Acceso a Persistencia / Public Stored Procedures / Private Stored Procedures *[Capas de Nilsson]*

Evans propone una arquitectura muy parecida a la de Fowler, pero sin capa de *Acceso al Negocio* o *[Service Layer](http://martinfowler.com/eaaCatalog/serviceLayer.html)*:



Hay un montón de formas de dividir un sistema de software en capas, pero a través de la experiencia y la convención, la industria del desarrollo de software ha convergido en este patrón llamado **Layered Architecture**, el cual nos muestra que, por lo menos, vamos a contar con una capa de dominio, una capa de infraestructura, una capa de aplicación y algún cliente o interfaz de usuario que consuma esa aplicación. Los nombres de las capas en las diferentes arquitecturas pueden variar, alguna capa puede estar o no estar, o se puede dividir en dos o más, pero lo importante es que **nunca puede faltar la capa de Dominio o de Negocio**. Esta capa se debe aislar del resto de las capas del sistema. **Esta capa será la expresión final del negocio, la implementación limpia de un modelo guiado por el dominio.** Domain-Driven Design requiere sólo la capa de negocio para existir. **Los objetos de dominio deben quedar libres de la responsabilidad de mostrarse, guardarse y/o responder a tareas de la aplicación, para poder enfocarse en expresar el modelo de dominio.**

PATRONES DE DISENO:

Los **patrones de diseño** son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de [software](http://es.wikipedia.org/wiki/Software) y otros ámbitos referentes al diseño de interacción o interfaces.

Un patrón de diseño es una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su **efectividad** resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser **reutilizable**, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

**Patrones creacionales**

* [Object Pool](http://es.wikipedia.org/w/index.php?title=Object_Pool_(patr%C3%B3n_de_dise%C3%B1o)&action=edit&redlink=1) (no pertenece a los patrones especificados por GoF): se obtienen objetos nuevos a través de la clonación. Utilizado cuando el costo de crear una clase es mayor que el de clonarla. Especialmente con objetos muy complejos. Se especifica un tipo de objeto a crear y se utiliza una interfaz del prototipo para crear un nuevo objeto por clonación. El proceso de clonación se inicia instanciando un tipo de objeto de la clase que queremos clonar.
* [Abstract Factory](http://es.wikipedia.org/wiki/Abstract_Factory_(patr%C3%B3n_de_dise%C3%B1o)) (fábrica abstracta): permite trabajar con objetos de distintas familias de manera que las familias no se mezclen entre sí y haciendo transparente el tipo de familia concreta que se esté usando.
* [Builder](http://es.wikipedia.org/wiki/Builder_(patr%C3%B3n_de_dise%C3%B1o)) (constructor virtual): abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto.
* [Factory Method](http://es.wikipedia.org/wiki/Factory_Method_(patr%C3%B3n_de_dise%C3%B1o)) (método de fabricación): centraliza en una clase constructora la creación de objetos de un subtipo de un tipo determinado, ocultando al usuario la casuística para elegir el subtipo que crear.
* [Prototype](http://es.wikipedia.org/wiki/Prototype_(patr%C3%B3n_de_dise%C3%B1o)) (prototipo): crea nuevos objetos clonándolos de una instancia ya existente.
* [Singleton](http://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o_Singleton) (instancia única): garantiza la existencia de una única instancia para una clase y la creación de un mecanismo de acceso global a dicha instancia.

**Patrones estructurales**

* [Adapter](http://es.wikipedia.org/wiki/Adapter_(patr%C3%B3n_de_dise%C3%B1o)) (Adaptador): Adapta una interfaz para que pueda ser utilizada por una clase que de otro modo no podría utilizarla.
* [Bridge](http://es.wikipedia.org/wiki/Bridge_(patr%C3%B3n_de_dise%C3%B1o)) (Puente): Desacopla una abstracción de su implementación.
* [Composite](http://es.wikipedia.org/wiki/Composite_(patr%C3%B3n_de_dise%C3%B1o)) (Objeto compuesto): Permite tratar objetos compuestos como si de uno simple se tratase.
* [Decorator](http://es.wikipedia.org/wiki/Decorator_(patr%C3%B3n_de_dise%C3%B1o)) (Envoltorio): Añade funcionalidad a una clase dinámicamente.
* [Facade](http://es.wikipedia.org/wiki/Facade_(patr%C3%B3n_de_dise%C3%B1o)) (Fachada): Provee de una interfaz unificada simple para acceder a una interfaz o grupo de interfaces de un subsistema.
* [Flyweight](http://es.wikipedia.org/wiki/Flyweight_(patr%C3%B3n_de_dise%C3%B1o)) (Peso ligero): Reduce la redundancia cuando gran cantidad de objetos que poseen idéntica información.
* [Proxy](http://es.wikipedia.org/wiki/Proxy_(patr%C3%B3n_de_dise%C3%B1o)): Mantiene un representante de un objeto.
* [Modulo](http://es.wikipedia.org/wiki/Module_(patr%C3%B3n_de_dise%C3%B1o)): Agrupa varios elementos relacionados, como clases, singletons, y metodos, utilizados globalmente, en una entidad unica.

**Patrones de comportamiento**

* [Chain of Responsibility](http://es.wikipedia.org/wiki/Chain_of_Responsibility_(patr%C3%B3n_de_dise%C3%B1o)) (Cadena de responsabilidad): Permite establecer la línea que deben llevar los mensajes para que los objetos realicen la tarea indicada.
* [Command](http://es.wikipedia.org/wiki/Command_(patr%C3%B3n_de_dise%C3%B1o)) (Orden): Encapsula una operación en un objeto, permitiendo ejecutar dicha operación sin necesidad de conocer el contenido de la misma.
* [Interpreter](http://es.wikipedia.org/wiki/Interpreter_(patr%C3%B3n_de_dise%C3%B1o)) (Intérprete): Dado un lenguaje, define una gramática para dicho lenguaje, así como las herramientas necesarias para interpretarlo.
* [Iterator](http://es.wikipedia.org/wiki/Iterator_(patr%C3%B3n_de_dise%C3%B1o)) (Iterador): Permite realizar recorridos sobre objetos compuestos independientemente de la implementación de estos.
* [Mediator](http://es.wikipedia.org/wiki/Mediator_(patr%C3%B3n_de_dise%C3%B1o)) (Mediador): Define un objeto que coordine la comunicación entre objetos de distintas clases, pero que funcionan como un conjunto.
* [Memento](http://es.wikipedia.org/wiki/Memento_(patr%C3%B3n_de_dise%C3%B1o)) (Recuerdo): Permite volver a estados anteriores del sistema.
* [Observer](http://es.wikipedia.org/wiki/Observer_(patr%C3%B3n_de_dise%C3%B1o)) (Observador): Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y actualicen automáticamente todos los objetos que dependen de él.
* [State](http://es.wikipedia.org/wiki/State_(patr%C3%B3n_de_dise%C3%B1o)) (Estado): Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.
* [Strategy](http://es.wikipedia.org/wiki/Strategy_(patr%C3%B3n_de_dise%C3%B1o)) (Estrategia): Permite disponer de varios métodos para resolver un problema y elegir cuál utilizar en tiempo de ejecución.
* [Template Method](http://es.wikipedia.org/wiki/Template_Method_(patr%C3%B3n_de_dise%C3%B1o)) (Método plantilla): Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos, esto permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.
* [Visitor](http://es.wikipedia.org/wiki/Visitor_(patr%C3%B3n_de_dise%C3%B1o)) (Visitante): Permite definir nuevas operaciones sobre una jerarquía de clases sin modificar las clases sobre las que opera.

**Patrones de interacción**

El primer intento por aplicar este concepto en el diseño de las interfaces de usuario se dio por Ward Cummingham y Kent Beck quienes adaptaron la propuesta de C. Alexander y crearon cinco patrones de interfaz: *Window per task*, *Few panes*, *Standard panes*, *Nouns and verbs*, y *Short Menu*. En años más recientes investigadores como Martin Van Welie, Jennifer Tidwell, Jaime Muñoz han desarrollado colecciones de patrones de interacción para la [World Wide Web](http://es.wikipedia.org/wiki/World_Wide_Web" \o "World Wide Web). En dichas colecciones captan la experiencia de programadores y diseñadores expertos en el desarrollo de interfaces usables y condensan esta experiencia en una serie de guías o recomendaciones, que puedan ser usadas por los desarrolladores novatos con el propósito de que en poco tiempo adquieran la habilidad de diseñar interfaces que incidan en la satisfacción de los usuarios. Los patrones de interacción buscan la reutilización de interfaces eficaces y un manejo óptimo de los recursos de las [páginas web](http://es.wikipedia.org/wiki/P%C3%A1gina_web), haciendo más eficaz el consumo de tiempo en el diseño del [sitio web](http://es.wikipedia.org/wiki/Sitio_web) y permitiendo a los programadores novatos adquirir más experiencia.

**TESTING**

**PRUEBAS DE CAJA NEGRA**

La prueba verifica que el ítem se está probando, cuando se dan las entradas apropiadas produce los resultados esperados. Los datos de prueba se escogerán atendiendo a las especificaciones del problema, sin importar los detalles internos del programa, a fin de verificar que el programa corra bien.

El método de la caja negra se centra en los requisitos fundamentales del software y permite obtener entradas que prueben todos los requisitos funcionales del programa.

Con este equipo de pruebas se intenta encontrar:

• Funciones incorrectas o ausentes.

• Errores de interfaz.

• Errores en estructuras de datos o en accesos a las bases de datos externas.

• Errores de rendimiento.

• Errores de inicialización y terminación.

La idea básica es enumerar una lista de errores posibles o de situaciones propensas a error y después escribir los casos de prueba basados en la lista. Por ejemplo, la presencia del valor 0 en la entrada de un programa es una situación con tendencia a error. Por lo tanto, puede ser que se escriba los casos de prueba para los cuales los valores particulares de la entrada tienen valor 0 y para qué valores particulares de la salida se colocan de manera forzada a 0.

**PRUEBAS DE CAJA BLANCA**

Consiste en realizar pruebas para verificar que líneas específicas de código funcionan tal como está definido. También se le conoce como prueba de caja-transparente.

La prueba de la caja blanca es un método de diseño de casos de prueba que usa la estructura de control del diseño procedimental para derivar los casos de prueba.

Las pruebas de caja blanca intentan garantizar que:

• Se ejecutan al menos una vez todos los caminos independientes de cada módulo

• Se utilizan las decisiones en su parte verdadera y en su parte falsa

• Se ejecuten todos los bucles en sus límites

•Se utilizan todas las estructuras de datos internas.

Para esta prueba se consideran tres importantes puntos.

• Conocer el desarrollo interno del programa, determinante en el análisis de coherencia y consistencia del código.

• Considerar las reglas predefinidas por cada algoritmo.

• Comparar el desarrollo del programa en su código con la documentación pertinente.

**PRUEBAS UNITARIAS**

Estas pruebas las corre el desarrollador, cada vez que va probando pedazos de código o scripts para ver si todo funciona como el desea. Estas pruebas son muy técnicas. Ejemplo: probar una consulta, probar que un pedazo de código envié algo a imprimir, probar que una función devuelva un flag, etc.

Para que una prueba unitaria sea buena se deben cumplir los siguientes requisitos:

•Automatizable: no debería requerirse una intervención manual. Esto es especialmente útil para integración continúa.

•Completas: deben cubrir la mayor cantidad de código.

•Repetibles o Reutilizables: no se deben crear pruebas que sólo puedan ser ejecutadas una sola vez. También es útil para integración continua.

•Independientes: la ejecución de una prueba no debe afectar a la ejecución de otra.

•Profesionales: las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.

Ventajas

El objetivo de las pruebas unitarias es aislar cada parte del programa y mostrar que las partes individuales son correctas. Proporcionan un contrato escrito que el trozo de código debe satisfacer.

Limitaciones

Es importante darse cuenta de que las pruebas unitarias no descubrirán todos los errores del código. Por definición, sólo prueban las unidades por sí solas. Por lo tanto, no descubrirán errores de integración, problemas de rendimiento y otros problemas que afectan a todo el sistema en su conjunto. Además, puede no ser trivial anticipar todos los casos especiales de entradas que puede recibir en realidad la unidad de programa bajo estudio. Las pruebas unitarias sólo son efectivas si se usan en conjunto con otras pruebas de software.

**INYECCION DE DEPENDENCIA**

El patrón de inyección de dependencias consiste en hacer que nuestras piezas de software sean independientes comunicándose únicamente a través de un interface. Esto implica muchas modificaciones en el código fuente como el uso de implementaciones, la eliminación de la instanciación de objetos mediante la instrucción **new** o la necesidad de un modo de configuración que indique que clases se instanciarán en el caso de solicitarlo.

Con inyección de dependencias perdemos visión (no abstracta) en tiempo de diseño y con ello el datalle de lo que hace la aplicación. Además perdemos funcionalidades de depuración en módulos en proyectos externos ya que es preciso recompilar estos para poder modificarlos. Sin embargo, ganamos en la posibilidad de hacer **mocking** para el testeo automático y ganamos en la posibilidad de reemplazar piezas que cumplan un estandar.

La siguiente cuestión que habrá que plantearse, ¿se pueden definir estándares para establecer unos interfaces según el negocio y abrir un mercado de desarrollo de módulos?. Para que algo se convierta en estándar es preciso que sea muy utilizado.

Ahora mismo estamos en estado de experimentación aún definiendo como se tiene que hacer la inyección. Cuando ya sepamos como se debe hacer la inyección pasaremos a la siguiente etapa que es **la definición de interfaces estándares**.

**ORIENTACION A OBJETOS**

**Clases abstractas**

La abstracción es un recurso de la mente (quizás el más característico de nuestra pretendida superioridad respecto del mundo animal). Por su parte, los lenguajes de programación permiten expresar la solución de un problema de forma comprensible simultáneamente por la máquina y el humano. Constituyen un puente entre la abstracción de la mente y una serie de instrucciones ejecutables por un dispositivo electrónico. En consecuencia, la capacidad de abstracción es una característica deseable de los lenguajes artificiales, pues cuanto mayor sea, mayor será su aproximación al lado humano. Es decir, con la imagen existente en la mente del programador. En este sentido, la introducción de las clases en los lenguajes orientados a objetos ha representado un importante avance respecto de la programación tradicional y dentro de ellas, las denominadas clases abstractas son las que representan el mayor grado de abstracción.

De hecho, las clases abstractas presentan un nivel de "abstracción" tan elevado que no sirven para instanciar objetos de ellas. Representan los escalones más elevados de algunas jerarquías de clases y solo sirven para derivar otras clases, en las que se van implementando detalles y concreciones, hasta que finalmente presentan un nivel de definición suficiente que permita instanciar objetos concretos. Se suelen utilizar en aquellos casos en que se quiere que una serie de clases mantengan una cierta característica o interfaz común. Por esta razón a veces se dice de ellas que son pura interfaz.

**INTERFACES**

Una interfaz es una especie de plantilla para la construcción de clases. Normalmente una interfaz se compone de un conjunto de declaraciones de cabeceras de métodos (sin implementar, de forma similar a un método abstracto) que especifican un protocolo de comportamiento para una o varias clases. Además, una clase puede implementar una o varias interfaces:

en ese caso, la clase debe proporcionar la declaración y definición de todos los métodos de cada una de las interfaces o bien declararse como clase abstract. Por otro lado, una interfaz puede emplearse también para declarar constantes que luego puedan ser utilizadas por otras clases.

Una interfaz puede parecer similar a una clase abstracta, pero existen una serie de diferencias entre una interfaz y una clase abstracta:

 Todos los métodos de una interfaz se declaran implícitamente como abstractos y públicos.

 Una clase abstracta no puede implementar los métodos declarados como abstractos, una interfaz no puede implementar ningún método (ya que todos son abstractos).

 Una interfaz no declara variables de instancia.

 Una clase puede implementar varias interfaces, pero sólo puede tener una clase ascendiente directa.

 Una clase abstracta pertenece a una jerarquía de clases mientras que una interfaz no pertenece a una jerarquía de clases. En consecuencia, clases sin relación de herencia pueden implementar la misma interfaz

**SCRUM**

Scrum es un marco de trabajo para la gestión y desarrollo de software basada en un proceso iterativo e incremental utilizado comúnmente en entornos basados en el desarrollo ágil de software.

Scrum es un modelo de referencia que define un conjunto de prácticas y roles, y que puede tomarse como punto de partida para definir el proceso de desarrollo que se ejecutará durante un proyecto. Los roles principales en Scrum son el ScrumMaster, que mantiene los procesos y trabaja de forma similar al director de proyecto, el ProductOwner, que representa a los stakeholders (interesados externos o internos), y el Team que incluye a los desarrolladores.

Durante cada sprint, un periodo entre una y cuatro semanas (la magnitud es definida por el equipo), el equipo crea un incremento de software potencialmente entregable (utilizable). El conjunto de características que forma parte de cada sprint viene del Product Backlog, que es un conjunto de requisitos de alto nivel priorizados que definen el trabajo a realizar. Los elementos del Product Backlog que forman parte del sprint se determinan durante la reunión de Sprint Planning. Durante esta reunión, el Product Owner identifica los elementos del Product Backlog que quiere ver completados y los hace del conocimiento del equipo. Entonces, el equipo determina la cantidad de ese trabajo que puede comprometerse a completar durante el siguiente sprint.[2](http://es.wikipedia.org/wiki/Scrum#cite_note-schwaber-1) Durante el sprint, nadie puede cambiar el Sprint Backlog, lo que significa que los [requisitos](http://es.wikipedia.org/wiki/Requisitos) están congelados durante el sprint.

Scrum permite la creación de equipos autoorganizados impulsando la co-localización de todos los miembros del equipo, y la comunicación verbal entre todos los miembros y disciplinas involucrados en el proyecto.

Un principio clave de Scrum es el reconocimiento de que durante un proyecto los clientes pueden cambiar de idea sobre lo que quieren y necesitan (a menudo llamadorequirements churn), y que los desafíos impredecibles no pueden ser fácilmente enfrentados de una forma predictiva y planificada. Por lo tanto, Scrum adopta una aproximación pragmática, aceptando que el problema no puede ser completamente entendido o definido, y centrándose en maximizar la capacidad del equipo de entregar rápidamente y responder a requisitos emergentes.

Existen varias implementaciones de sistemas para gestionar el proceso de Scrum, que van desde notas amarillas "post-it" y pizarras hasta paquetes de software. Una de las mayores ventajas de Scrum es que es muy fácil de aprender, y requiere muy poco esfuerzo para comenzarse a utilizar.

**Documentos**

**Product backlog**

El product backlog es un documento de alto nivel para todo el proyecto. Contiene descripciones genéricas de todos los requerimientos, funcionalidades deseables, etc. priorizadas según su retorno sobre la inversión ([ROI](http://es.wikipedia.org/wiki/Retorno_de_la_inversi%C3%B3n)) . Es el qué va a ser construido. Es abierto y cualquiera puede modificarlo. Contiene estimaciones [grosso modo](http://es.wikipedia.org/w/index.php?title=Grosso_modo&action=edit&redlink=1), tanto del valor para el negocio, como del esfuerzo de desarrollo requerido. Esta estimación ayuda al product owner a ajustar la línea temporal y, de manera limitada, la prioridad de las diferentes tareas. Por ejemplo, si dos características tienen el mismo valor de negocio la que requiera menor tiempo de desarrollo tendrá probablemente más prioridad, debido a que su [ROI](http://es.wikipedia.org/wiki/Retorno_de_la_inversi%C3%B3n" \o "Retorno de la inversión)será más alto.

**Sprint backlog**

El sprint backlog es un documento detallado donde se describe el cómo el equipo va a implementar los requisitos durante el siguiente sprint. Las tareas se dividen en horas con ninguna tarea de duración superior a 16 horas. Si una tarea es mayor de 16 horas, deberá ser divida en otras menores. Las tareas en el sprint backlog nunca son asignadas, son tomadas por los miembros del equipo del modo que les parezca oportuno.