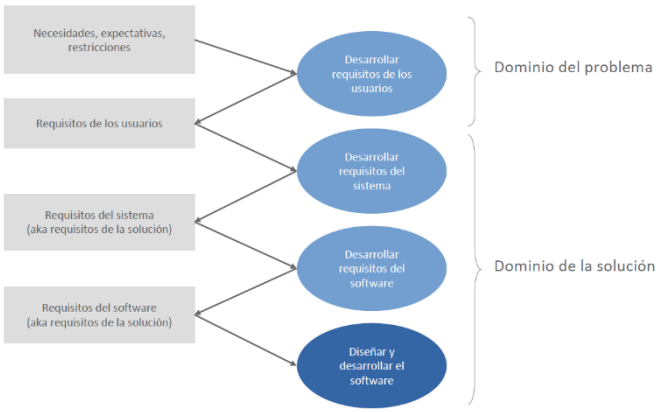
Introducción al diseño de software

Tabla de contenido

Introducción

Hasta ahora lo que nosotros hemos ido viendo es que a partir de las necesidades, nosotros hacemos un desarrollo de requisitos que da como resultado una especificación de requisitos de los usuarios que a su vez eso sirve como base para desarrollar los requisitos del sistema que tenemos que construir. De alguna manera los requisitos del usuario son unos requisitos del tipo "caja negra", los requisitos del sistema nos permiten ver un poquito más los aspectos internos, no de implementación, sino que tienen que ver con el comportamiento del sistema y eso forma la base para poder desarrollar la solución, diseñar y desarrollar las soluciones, diseñar y desarrollar el software.



Diseño de software

Definición

El diseño software es el proceso mediante el cual definimos la arquitectura, los componentes que forman parte de la arquitectura, las interfaces entre esos elementos, y por supuesto, entre el sistema y el mundo exterior. Y adicionalmente, es lo que permite este definir algunas otras características de los sistemas y de las componentes. También, diseño de software es el resultado del proceso, con lo cual, cuando usamos el término diseño de software, nos vamos a estar definiendo probablemente a estas 2 acepciones.

Fundamentos

- El diseño software fundamentalmente es una actividad creativa, no hay una receta para diseñar, hay lineamientos, hay patrones, pero también tiene que ver mucho el "buen gusto", si se quiere, al diseñar (dar buenos nombres a las variables, dar algunos nombres a las clases, seguir estándares), pero de todas maneras hay una actividad profundamente creativa ahí, que implica trasladar lo que tenemos definido en el mundo de requisitos, al mundo restringido del diseño.

- Implica indefectiblemente tomar decisiones. Siempre que miramos el código escrito por otro nos preguntamos "bueno, ¿por qué acá hizo esto? podría haberlo hecho de esta otra manera", bueno, eso tiene que ver con decisiones que tienen que ver con las circunstancias que nos rodean al momento de hacer diseño/construir, con lo cual el diseño implica que tomemos decisiones. Algunas van a tener que ver con salir del paso y poder entregar rápidamente y otras que tendrán que ver más con decisiones a largo plazo.

- El diseño de alguna manera también es emergente. ¿Se puede hacer diseño upfront? Por supuesto. Lo que no podemos estar es 6 meses haciendo diseño upfront. De alguna manera, como hemos visto a lo largo de todo el curso, las cosas son naturalmente iterativas, incrementales y el diseño de alguna manera va a ir tomando forma, va a ir emergiendo. Por supuesto que algo de arquitectura hay que pensar antes de salir o mandarse a diseñar y a desarrollar. Por supuesto que hay que tomar grandes decisiones.

- Por supuesto que diseñar no es sinónimo de modelar. Hemos hablado hasta el hartazgo de que para este tipo de ingeniería tan particular en la que estamos nosotros que es la ingeniería de software, el código es un modelo más, digamos, cuando estamos programando en realidad estamos de alguna manera diseñando, no existe la construcción en el sentido clásico. En algún momento, sobre todo fines de los 90 principios de este siglo, estaba la idea de que diseñar era modelar. De hecho, yo he tenido reuniones con algunos clientes que me han llegado a decir cosas tales como "bueno, nosotros hacemos arquitectura, solamente hacemos dibujitos", lo cual es una barbaridad. Digamos, no es así. Diseñar implica, en algunos casos hacer modelo, pero no es estrictamente únicamente modelar el diseño, sino que implica también, meterse en el código, hacer un análisis detallado de eso, etc.

[**V** / F] Cuando se refiere al **Diseño de Software** es válido tanto para el proceso mediante el cual se define la arquitectura, las interfaces y los componentes del sistema como para los resultados de dicho proceso.

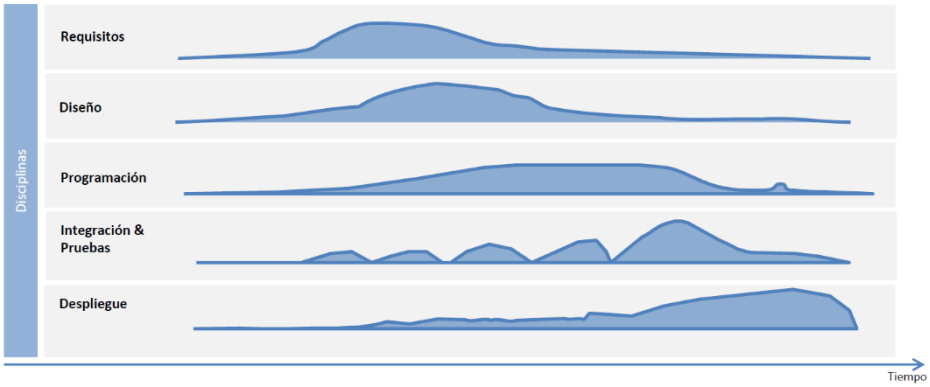
Modelado: Dos caminos posibles

Con respecto al modelado, de todas maneras, como ustedes ya saben, hay 2 caminos posibles:

- **Modelado ágil**: Está la escuela que nos dice, bueno, en realidad, el modelado es una actividad que hacemos tanto en los requisitos como en los diseño, para poder entender un poco mejor, para poder definir un poco mejor.
  - *Modelos informales*: No se suelen reutilizar, se pueden hacer en una pizarra, no hace falta invertir mucho esfuerzo en documentarlo.
  - *Costo de mantener los modelos Vs. utilidad de los modelos*.
- **Ingeniería de software basada en modelos o model-driven engineering**: Esta escuela tiene que ver con fundamentalmente desarrollar modelos, transformar esos modelos finalmente en el producto final, utilizando una serie de herramientas muy, muy poderosas.
  - *Los modelos evolucionan y se transforman en el software*: Esta era la idea de las herramientas case de los años 90 que parcialmente cumplieron su cometido pero que, a medida que se fue complicando la tecnología, a medida que fueron apareciendo aplicaciones cliente-servidor, fueron apareciendo aplicaciones web, etc., se fue transformando en una actividad demasiado compleja.
  - *Relativamente popular en determinados* segmentos: Hay unos ciertos segmentos (muy particulares) de la industria que están utilizando este camino. La idea es que uno pueda desarrollar un modelo de requisitos; a partir de ese modelo de requisitos y con algunas transformaciones, generar un modelo de diseño; y a través de unas transformaciones, a partir de ese modelo de diseño, generar el código, poder hacer la prueba y poner en producción. Eso sería el sueño de cualquier desarrollador de software, pero lo cierto es que la tecnología detrás de eso es muy complicada, muy compleja. Y lo que hemos visto, lo que en general se comenta en la industria es que no es algo que se haya masificado. Sin embargo, hay algunos segmentos muy particulares en la industria en donde este tipo de cosas sí se utiliza.

De los requisitos al diseño y la construcción

Una transición iterativa e incremental



¿Cómo hacemos para transformar los requisitos en diseño? Bueno, no es un camino fácil. Ustedes saben que este, las disciplinas del desarrollo de software se superponen, ahí con. Es cierto, grado de intensidad de acuerdo a donde hemos parado claramente. Empezamos con los requisitos y con el diseño relativamente temprano en el ciclo de vida. Si ubicamos esas disciplinas a lo largo del tiempo vemos que uno tiene que empezar a diseñar un poco al comienzo del proyecto, lo que no, cual no quiere decir que tenga que diseñar todo, absolutamente todo, antes de trasladarse. Diseño del código, lo que tiene que pensar un poco. Tiene que tomar algunas decisiones de entrada, sobre todo con respecto a temas de arquitectura con con respecto a. A grandes si se quiere. Aspectos del sistema que uno va a desarrollar. Y, obviamente, la frontera entre requisitos y diseño no siempre está tan clara. Hay una sinergia bastante importante, hay tanto lo que les quiero decir, por ejemplo, que uno puede partir de un modelo de dominio.¿Es un modelo conceptual, es un modelo de de requisitos o del área de requisitos? Tiene que transformar eso en un diseño de base de Datos. Puede ser que eso también se utilice para diseñar o como base para diseñar las clases que tienen que ver con la lógica de la aplicación o con la parte más relacionada con el negocio, pero tranquilamente, a medida que uno se mete en el diseño pueden surgir nuevos aspectos relacionados con los requisitos que a lo mejor es importante trasladar el modelo de dominio. No sé, nos olvidamos de una entidad importante o de un atributo importante, con lo cual siempre hay ahí un ida y vuelta entre todas las disciplinas.

[V / **F**] Pasar de los requisitos al diseño es un camino relativamente fácil.

Nosotros hemos trabajado bastante hasta ahora en transformar las necesidades y las expectativas de los clientes, de los usuarios, en requisitos formalmente establecidos; hemos desarrollado a partir de esos requisitos del usuario, requisitos de software; hemos utilizado historias de usuario complementadas con criterios de aceptación. Bueno, de ahí deberíamos partir para poder definir cuál es la arquitectura del sistema que vamos a desarrollar, básicamente cuál es su organización, cuál es su estructura. A partir de ahí, también deberíamos poder identificar cuáles son las componentes, las clases, los algoritmos, las estructuras de datos internas, los métodos, etc. También a partir de esta definición deberíamos poder empezar a identificar cuáles son las interfaces de usuario. Ya hemos visto algo de eso cuando empezamos a trabajar sobre los prototipos. Ahí tenemos un primer intento de tratar de entender cuál podría ser la interfaz que la aplicación le ofrezca a nuestros usuarios. También hay datos que diseñar a partir de esta información. Y después está el tema de bueno, ver cómo hacen las distintas componentes, los distintos elementos para colaborar entre sí para resolver cada uno, de los escenarios de uso de la aplicación. Si estamos trabajando con historias de usuario, probablemente tengamos que determinar, también, cuáles son las distintas componentes que vamos a utilizar (de interfaz del usuario, de lógica de datos, etc.) para resolver esa historia de usuario.

Básicamente no hay una transición fácil entre los 2 mundos. No hay recetas como decimos siempre. No es lineal, pero hay algunas estrategias que se pueden utilizar.

Principios clásicos de diseño

Ustedes en materias anteriores han visto muchos principios clásicos que tienen que ver con el diseño del software:

- Abstracción
- Mantener una alta cohesión y un bajo acople
- Descomposición y modularización
- Encapsulamiento de la lógica y ocultamiento de la información: Tratar de adoptar estrategias para que los cambios que inevitablemente va a haber que hacer en el código queden circunscriptos a determinadas áreas del código.
- Separación de interfaz e implementación
- Separación de intereses
- Concurrencia de procesos
- Control y manejo de eventos
- Persistencia de datos
- Distribución de componentes
- Manejo de errores y excepciones
- Tolerancia a fallas
- Interacción y presentación
- Seguridad

Entonces ahí unos principios clásicos de diseño que ustedes ya conocen, que por ahí es importante darles una mirada, sobre todo no tanto desde el punto de vista de la visión más de las materias de programación, sino más bien

de una perspectiva, si se quiere, más mirando esto desde el punto de vista del diseño de grandes sistemas. Como decimos siempre, una cosa es hacer un pequeño desarrollo entre un pequeño equipo de personas y otra cosa completamente distinta es desarrollar software, como decimos nosotros, "in the large", o sea, desarrollar grandes productos de software, grandes productos con varios millones de líneas de código, claramente ahí, si uno no es ordenado y si no se pone de acuerdo en los principios de diseño que va a aplicar, va a terminar teniendo un código imposible de comprender.

Principios clásicos del diseño de objetos

GRASP: General Responsability Assignment Software Patterns (Principles)

Hay algunos principios que propone Craig Larman que recomendamos que revisen, que tienen que ver con los patrones o principios de asignación de responsabilidades. Larman hace bastantes recomendaciones con respecto a cómo asignar responsabilidades a distintos tipos de objetos:

- Controller
- Creator
- Indirection
- Information expert
- High cohesion
- Low coupling
- Polymorphism
- Protected variations
- Pure fabrication

SOLID

- **S**ingle-responsability
- **O**pen-Closed
- **L**ikov substitution
- **I**nterface segregation
- **D**ependency inversion

Patrones de diseño

Gang of four (GoF)

Patrones que son complementarios a los que ustedes ya conocen, que son por ahí conocidos como "Gang of Four" que tiene que ver con cómo creamos objetos, cómo estructuramos el diseño y cómo manejamos el comportamiento:

- **Creational Design Patterns**: The design patterns that deal with the creation of an object.
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton

- **Structural Design Patterns**: The design patterns in this category deals with the class structure such as Inheritance and Composition.
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy

- **Behavior Design Patterns**: This type of design patterns provide solution for the better interaction between objects, how to provide lose coupling, and flexibility to extend easily in future.
  - Chain of responsability
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template method
  - Visitor

Algunos de los principios clásicos de diseño de software son:

- ☑ **Polimorfismo (GRASP)**
- ☑ **Abstracción**
- ☑ **Creator (GRASP)**
- ☑ **Inversión de dependencias (SOLID)**
- ☑ **Concurrencia**
- ☑ **Singleton (Gang of Four)**

Arquitectura

Definición

Hablamos recién de las distintas dimensiones que tiene el diseño, hablamos de los temas de arquitectura. Claramente, como hemos visto con anterioridad, la **arquitectura de software** es la estructura de alto nivel que tiene un sistema de software, la disciplina de creación de este tipo de estructuras y la documentación de dichas estructuras.

Más definiciones

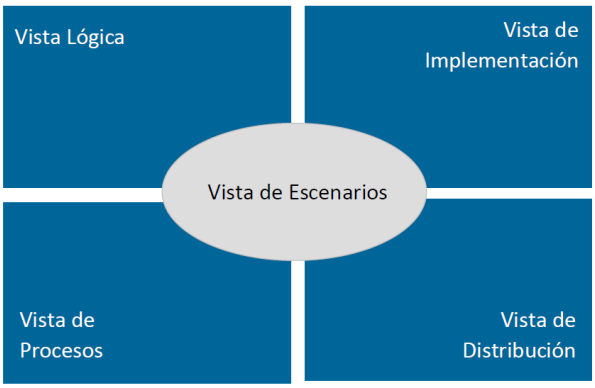
Cuando hablamos de arquitectura de alto nivel nos relacionamos con temas que tienen que ver con las grandes decisiones que se toman con respecto a la estructura del sistema, decisiones que, como dice algún autor por ahí, van a ser difíciles de cambiar:

- **Bass**: Es la estructura de un sistema. Comprende a las componentes del software, a las propiedades externamente visibles de dichas componentes y a las relaciones entre ellas.
- **Shaw**: Es la descripción de los elementos que forman parte de un sistema, las interacciones entre dichos elementos, el patrón que rige su organización, y las restricciones que limitan a ese patrón.
- **Booch**: Comprende al conjunto de decisiones significativas respecto a la organización del sistema: la selección de elementos estructurales y sus interfaces, el comportamiento de los elementos, la composición progresiva en estructuras mayores, y el estilo que guía la organización.

Si se quiere, este es el diseño de alto nivel como llaman algunos. Aunque quizás sea un término incorrecto porque alto nivel de la idea de que no profundizamos en detalles y lo cierto es que arquitectura también tiene que ver con algunos temas de detalles que son importantes. Con lo cual, quizás, hay que tener cuidado con los términos que utilizamos

4 vistas + 1

Hay un modelo para entender cómo es la arquitectura llamado "4 vistas + 1" (que fue propuesto por Philippe Kruchten hace ya más de 20 años), que lo que nos plantea es que, como hemos visto al leer a Frederick Brooks[1], el software no es puente, el software no es una pieza material, entonces intentar representar a través de los modelos una realidad que no existe en el espacio es un poco complicado. Entonces, lo que plantea Kruchten es que uno podría haber un producto de software / aplicación / sistema basado en software utilizando varias vistas que nos van a mostrar aspectos particulares de cada una de las dimensiones que forman parte del software:



- Una tiene que ver, si se quiere, con los aspectos lógicos (**vista lógica**). La estructura lógica que tiene el sistema en función de subsistemas, componentes, interfaces entre los sistemas, entre los subsistemas, etc. Si se quiere también una estructura que tiene que ver con, siguiendo el diseño orientado a

[1] En referencia al texto de No Silver Bullet.

objetos, las clases que forman parte de la definición que hemos hecho.

- La **vista de implementación** tiene más que ver con cómo empaquetamos esos elementos en componentes ejecutables, en componentes que son manipulables por el entorno de producción, por el entorno que vamos a utilizar para explotar el producto. En definitiva, esa vista lógica que describe la estructura del código en clases, métodos, etc. va a ser empaquetada en ejecutables. Algunos de esos ejecutables van a estar en algún servidor, algunos otros van a estar en otro lugar, bueno eso hay que verlo, hay que entenderlo. Eso también se muestra parcialmente en la vista en la **vista de distribución**.
- La **vista del proceso** tiene que ver también con los procesos que se van a estar ejecutando a nivel del sistema operativo.

Por supuesto que todas estas vistas no son disjuntas. Por supuesto que no va a ser falta que desarrollemos todas estas vistas para todos los sistemas de construyamos, pero es un punto bastante interesante. La vista lógica la podemos representar con un diagrama de clases, por ejemplo; la vista de distribución podemos utilizar un modelo de UML que nos muestre los nodos de procesamiento; la vista de procesos también tiene su propia notación; la vista de implementación nos puede mostrar ya componentes físicos como un ejecutable, un DLL[2], una librería o lo que sea. Lo que es interesante de lo que plantea Kruchten es que hay una especie de vista que junta todo esto, que es la **vista de escenarios**. Cuando Kruchten plantea esto, lo plantea fundamentalmente pensando, sobre todo, en casos de uso, que es la forma que había de representar escenarios por la época en la que él escribe este paper, que si no recuerdo mal fue en la segunda mitad de los 90. En definitiva, uno podría decir agarrar y decir… bueno, a ver, independientemente de la estructura de clases que uno tenga en la implementación de este escenario particular, de esta historia de usuario en particular, las clases que van a participar son estas. Con lo cual nos puede mostrar acá una vista un poco más dinámica, y el mecanismo para mostrar esta vista dinámica es ni más ni menos que un diagrama de secuencia o un diagrama de colaboración.

Diseño arquitectónico y asignación/distribución de requisitos

Claramente hay que decidir qué modelos construir, si vale la pena construir los modelos de todas estas dimensiones, de todas estas vistas. Hay un costo implícito de mantener esas vistas vigentes durante el ciclo de vida completo del software, son decisiones. Probablemente cuando estamos desarrollando, necesitemos construir algunos de esos modelos. Probablemente durante el mantenimiento, durante la evolución, esos modelos en algún momento se pierdan, ya no tengan utilidad. Lo que sí es importante tener en cuenta es que, de alguna manera, la arquitectura va a influenciar en la arquitectura, los requisitos van a influenciar en la interfaz del usuario, los requisitos van a terminar impactando en las decisiones de diseño que tomemos. Ya en algún momento hemos hablado de este tema, sobre todo en la relación entre los requisitos no funcionales y las decisiones de arquitectura: Si uno quiere que sea mantenible, la arquitectura va a tener determinados atributos; si se quiere que sea muy segura/robusta, probablemente los atributos de performance se vean un poco perjudicados. Entonces hay, como de costumbre, hay una solución de compromiso en todo esto.

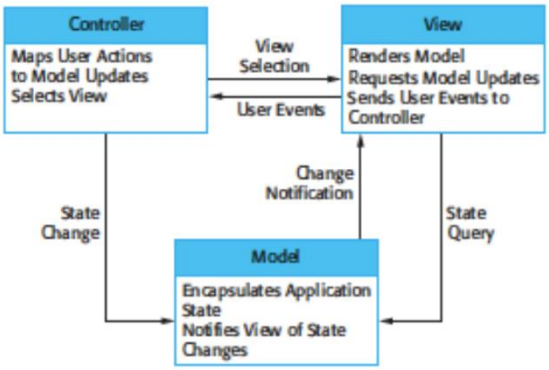
[**V** / F] Los requisitos influyen en las decisiones de diseño que se tomen y por lo tanto influyen en la arquitectura del sistema.

[2] Dynamic Link Library (Bibliotecas de Enlaces Dinámicos)

Estilos arquitectónicos

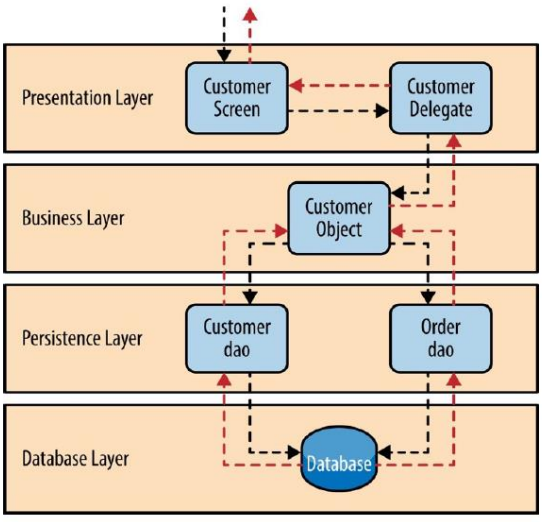
Hay estilos arquitectónicos que podemos revisar:

Model View Controller (MVC)



Un estilo arquitectónico que por ahí ya ustedes conocen por haberlo utilizado en alguna otra materia se conoce como Model View Controller. Este modelo lo que pretende es poder aislar los cambios en la interfaz del usuario de los cambios en la lógica de los cambios en la manera en la cual se persisten los datos. Entonces, en definitiva, lo que plantea el MVC es que la lógica relacionada con el diálogo con el usuario esté agrupada justamente en un conjunto de componentes que se conocen como "vista" o "View". Todo lo que tenga que ver con la lógica de negocio, con la persistencia de datos, con la estructura de los datos, está en "Modelo". Y el que se encarga de relacionar estos 2 elementos es algo llamado "Controller". Entonces, cuando a nosotros se nos muestra una página web, en definitiva, ahí están interviniendo componentes que tienen que ver con la vista; cuando yo completo un dato y hago un clic, el Controller va a estar recibiendo ese evento iba a poder relacionar ese click con la lógica de negocio o la lógica de la aplicación que va a estar registrada/especificada/detallada en el modelo. Entonces yo podría tener una aplicación con la lógica y con los datos persistidos a través del modelo (de lo que llamamos justamente en este esquema/estilo, la capa o el componente que tiene que ver con el modelo) y, de repente, podría tener una vista para una página web, podría tener un cliente para un celular, para un dispositivo móvil, podría tener distintos tipos de dispositivos. La lógica sería exactamente la misma, no cambiaría. En las aplicaciones monolíticas, esto que yo les estoy contando, está todo mezclado. En las viejas aplicaciones, en plataformas que no eran cliente-servidor, todo esto estaba medio mezclado. No había una separación clara de responsabilidades.

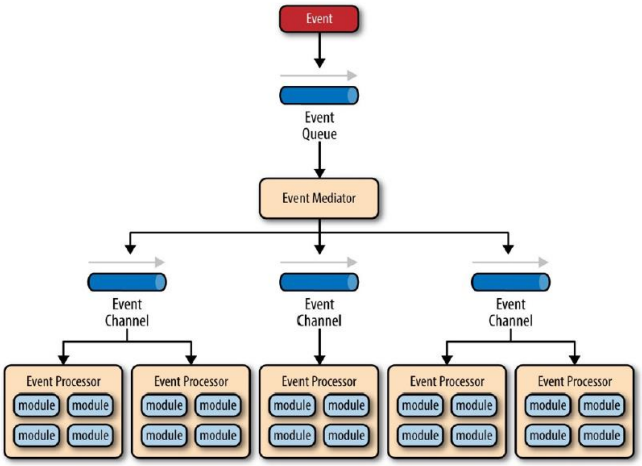
Layers



Otro tipo de estilo arquitectónico muy conocido es el de Capas. El de capas se parece un poco al que estábamos viendo recién. Lo que se plantea es que hay una **capa de presentación** que se encarga de manejar toda la lógica de diálogo con el usuario. Una capa que tiene que ver toda la lógica de la aplicación, por ejemplo, ¿qué pasa cuando un alumno consulta las materias puede cursar? Bueno, hay una lógica ahí que va a estar en esta **capa de negocio**. Y por supuesto, que va a haber que ir a buscar información a través de la **capa de persistencia** a una **base de datos**. Entonces, en definitiva, tengo acá la base de datos o los archivos que se acceden a través de una serie de componentes, clases o lo que fueren que van a estar acá en la capa de persistencia; La capa de la lógica de la aplicación, la lógica de negocio se va a encargar de tener, justamente como el nombre lo indica, todo lo que tenga que ver con el comportamiento que tiene que tener aplicación y vamos a delegar en esta capa todo lo que tenga que ver con la interfaz del usuario, de manera que si tenemos que hacer algún ajuste en algún elemento, claramente, vamos a minimizar los impactos en el resto. Por supuesto que ustedes pueden reconocer acá rápidamente, la aplicación en una escala un poco más grande de algunos de los principios de diseño que mencionábamos antes: el tema del mínimo acople y la máxima cohesión, la separación de responsabilidades, etc.

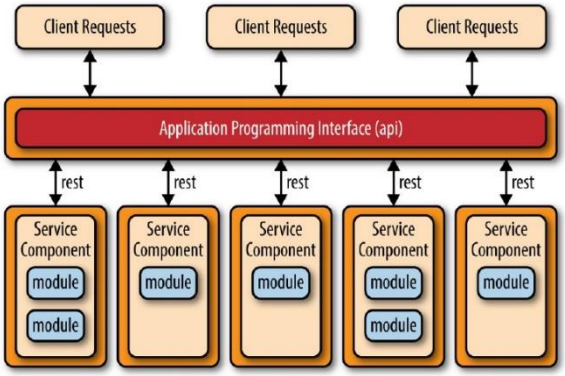
Event Driven (Basado en eventos)

Esto se utiliza también en ámbitos muy, muy específicos. Uno, en definitiva, publica un evento en una plataforma determinada; esa plataforma o cola de eventos se queda ahí escuchando, y del otro lado tenemos componentes que están escuchando esa cola y que están esperando que aparezcan esos eventos; cuando aparecen esos eventos se hacen determinadas cosas. Esto claramente no es algo que uno utilizaría para una aplicación interactiva, pero en definitiva, esto es lo que se conoce como estilo arquitectónico manejado por eventos o "Event Driven".



Microservicios

Lo que está hoy muy de moda y que van a utilizar ustedes en el trabajo práctico, es la arquitectura de microservicios, en donde uno tiene, a través de una interfaz llamada API, uno puede acceder a través de cierta estructura estandarizada de intercambio de mensajes puede acceder a lógica que está en una capa más abajo.



Objetivo

Cuando uno hace arquitectura lo que pretende es que, en una organización, todas las aplicaciones mantengan más o menos la misma arquitectura. ¿Para qué? Para poder conseguir la interoperabilidad entre distintas aplicaciones. La realidad es mucho más triste. La idea de la arquitectura, en términos generales, es que podamos determinar, podemos establecer que las aplicaciones tengan determinados tipos de arquitectura y que puedan interactuar entre sí. Que si yo necesito algo de otra arquitectura haya una interfaz definida común establecida para poder hacer esto.

Ejemplos de estilos arquitectónicos son:

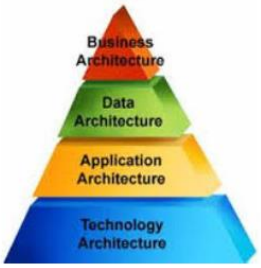
- ☐ 4+1[3]
- ☐ DAO (Data Access Object)[4]
- ☑ **Modelo Vista Controlador (MVC)**
- ☑ **Microservicios**
- ☑ **Capas (Layers)**
- ☑ **Manejado por Eventos**

[3] Es un modelo para entender cómo es la arquitectura.

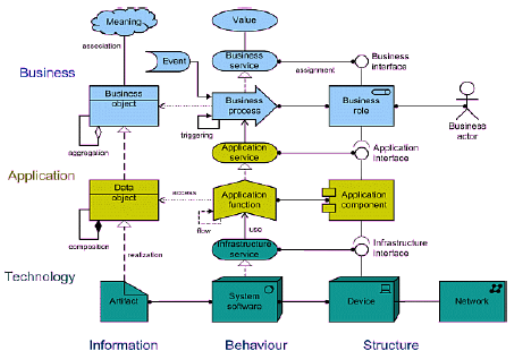
[4] Es un componente de software que suministra una interfaz común entre la aplicación y uno o más dispositivos de almacenamiento de datos , tales como una Base de datos o un archivo. [Wikipedia]

Arquitectura

Mas allá del software



Por supuesto que el tema no termina ahí. Uno podría pensar que cualquier organización necesita estandarizar no solamente los temas vinculados al software, sino también a lo que está por debajo del software, a los datos; y por supuesto, también lo que tiene que ver con la infraestructura. Entonces uno podría pensar en algo llamado por el nombre en inglés Entreprise Architecture o **arquitectura corporativa** (empresarial suena feo, por ahí arquitectura del negocio), en donde uno lo que termina de entender es… bueno, a ver: Yo soy una empresa que vende insumos para pymes. Tengo una serie de procesos que sigo: cómo vendo, cómo facturo, cómo entrego los pedidos… Bueno, eso está soportado con un conjunto de aplicaciones (**arquitectura aplicativa**). Y, a su vez, esas aplicaciones usan datos (**arquitectura de datos**) y están montadas sobre una infraestructura tecnológica (**arquitectura técnica**). Bueno, la idea de la arquitectura en un término un poco más amplio, implica poder entender esto, entender cuáles son los procesos clave, cómo utilizan las aplicaciones y cómo las aplicaciones hacen uso de la tecnología que está por detrás. Esto no lo ven ustedes, probablemente porque no tienen tanta experiencia, pero les puedo asegurar que en las grandes empresas, esto es un tema absolutamente clave.



Un tema que por ahí después vamos a comentar, y vuelvo un poco para atrás, al tema de arquitectura de software. Por ahí esto les parece demasiado abstracto, pero hay industrias que ya tienen definidas arquitecturas tipo. Esto está muy desarrollado en Europa, está muy desarrollado en EE. UU. Por ejemplo, hay iniciativas en Europa para desarrollar la interoperabilidad entre bancos. Casualmente en estos días acaba de salir una aplicación para tener acceso a las cuentas desde tu celular, acceso a las cuentas que cada uno tenemos en distintos bancos. Hoy eso, hasta ahora, era propietario. Ahora para poder acceder vos desde una aplicación a las cuentas de los diferentes bancos (que son de tu propiedad por supuesto) hace falta tener una arquitectura estandarizada entre los distintos actores. Bueno, eso es un ejemplo claro de una arquitectura, si se quiere, consensuada en una industria determinada. En Europa existe algo llamado “Open banking” que es ya algo bastante habitual, y en la industria de las telecomunicaciones existe algo llamado “eTOM” (enhanced Telecomunication Operations Map) que es una arquitectura estándar, si se quiere conceptual, que te dice cómo organizar las distintas piezas que forman parte de un negocio relativamente estándar que es el de las telecomunicaciones, y en esa bolsa entran no solamente lo que es telefonía fija, telefonía celular, sino también todo lo que tiene que ver con los cable operadores y ese tipo de cosas. Entonces, ¿a qué apunta esto? A que vos puedas intercambiar una pieza de tu arquitectura de manera un poco más fácil.

Interfaces

Diseño de la interacción entre los usuarios y el producto

Otro de los elementos que forman parte del diseño es el de las interfaces, las interfaces del usuario. Hay que definir la forma en la cual los usuarios van a interactuar con el cliente. Entonces, la verdad, es que en muchos de los casos es definitorio este tema. Podemos tener una muy buena aplicación performante o bien estructurada con una buena lógica, etc. pero si la interfaz es fea/mala, en consiguiente la/el aplicación/producto va a ser malo. Ya vamos a profundizar este tema en futuros capítulos, pero hay que tomar en cuenta que la interfaz termina siendo un elemento clave de la arquitectura. A continuación mencionamos algunos principios:

- Cuando hacemos cosas parecidas, las tenemos que hacer de la misma forma.
- La arquitectura de la interfaz de usuario nos permitiría poder ubicar la información fácilmente.
- Tiene que ser consistente.
- Nos tiene que servir a nosotros, que somos usuarios avanzados, pero también a los usuarios primerizos.
- Me tendría que permitir aprender

Todos estos elementos que parecen, medio si se quiere simplemente una enumeración de deseos, la verdad que es lo que uno espera y lo que espera un usuario de cualquier interfaz. Con lo cual, si no tomamos en cuenta estos temas que entran dentro de la categoría de requisitos no funcionales si se quieren, tienen un profundo impacto en el diseño y hay que tomarlo en cuenta.

Datos

Diseño de archivos y base de datos

También un tema que no hay que minimizar es el diseño de datos. Nosotros tenemos que definir cómo van a persistir los datos. Y este es un tema no menor, lamentablemente uno de los problemas que nosotros vemos recurrentemente es que algo muy positivo, muy bueno, como es la orientación a objetos, a veces, hace que muchos de los alumnos vengan con una idea, no del todo acabada de qué es lo que implica diseñar datos. No es algo que simplemente resuelve alguna capa en arquitectura, sino que hay que ponerse a trabajar ahí: Hay que definir, no solamente cuáles son las tablas, sino también hay que definir cuáles son los índices; Hay que ver en qué momento va a convenir normalizar o desnormalizar la base de datos… O sea, hay un montón de temas que no son tan transparentes.

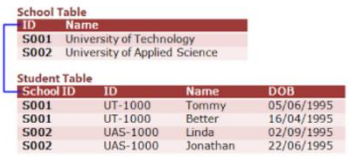
Tres niveles de modelado

Algunos entornos de desarrollo nos ayudan a resolver los problemas, pero lo cierto es que acá hay trabajo para hacer. A partir de un **modelo conceptual** como es el modelo de dominio, nosotros debemos poder definir el **diseño lógico** de la base de datos o el diseño lógico de los archivos, y a partir de ahí también hacer un **diseño físico**. He sido testigo de innumerables peleas entre programadores y administradores de bases de datos por las decisiones que los programadores toman sin saber lo que los administradores de base de datos están mirando. Con lo cual, esto es un tema que ustedes van a ver expandido con mucho más nivel de detalle en la materia correspondiente. Pero bueno, simplemente tomen en cuenta que el aspecto de datos es un elemento más que va a haber que tomar en consideración.

[V / **F**] Dentro de las actividades del diseño de software es más relevante lo establecido en la arquitectura, que el diseño de las interfaces de usuario o el diseño de datos

Bases de datos relacionales

Seguramente vamos a tener que definir cuáles son las **claves primarias** de las tablas que nos van a definir unívocamente a cada fila, a cada instancia. Las **claves foráneas** son lo que nos van a permitir implementar las asociaciones que vemos en el modelo de dominio (esto lo vamos a ver en más detalles en un capítulo posterior).

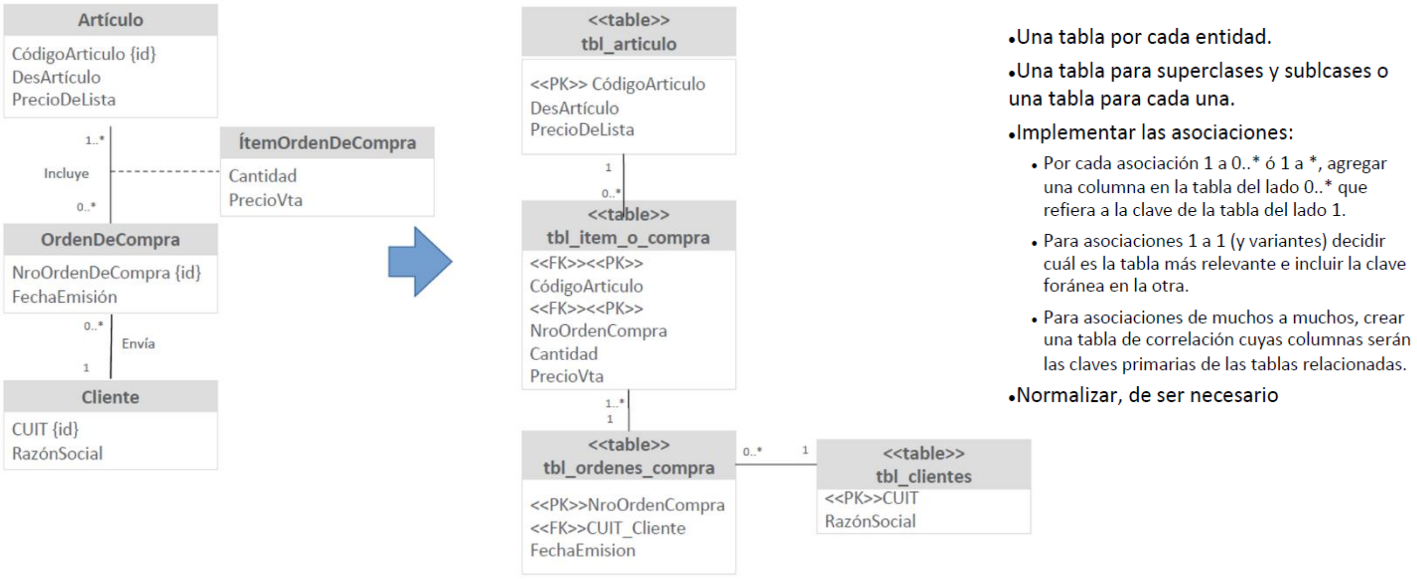

School Table

ID	Name
S001	University of Technology
S002	University of Applied Science

Student Table

School ID	ID	Name	DOB
S001	UT-1000	Tommy	05/06/1995
S001	UT-1000	Better	16/04/1995
S002	UAS-1000	Linda	02/09/1995
S002	UAS-1000	Jonathan	22/06/1995

Del modelo de dominio al modelo de base de datos relacional



Acá tenemos una transformación clásica del modelo de dominio al modelo, si se quiere, lógico de la base de datos. Entonces hay decisiones que tomar, hay que implementar este tipo de cosas, con lo cual nuevamente vuelvo al tema que el diseño tiene que ver con la arquitectura, tiene que ver con la interfaz del usuario y tiene que ver también con el tema de datos. Y por supuesto que hay una partecita del diseño que tiene que ver con, si se quiere, el diseño detallado. Y acá entramos en las posibles estrategias a adoptar para hacer el diseño.

Estrategias de diseño

Aquí tenemos distintas estrategias posibles:

- Diseño orientado a funciones
- Diseño orientado a objetos
- Diseño centrado en los datos
- Diseño basado en componentes
- Diseño orientado a aspectos
- Diseño orientado a servicios
- Diseño dirigido por el dominio (Domain-Driven Design)

El meollo de la cuestión es poder transformar ese modelo de análisis en para cada escenario, si se quiere, cuáles son las/los componentes/elementos que van a participar para resolver ese escenario, para resolver esa historia de usuario. Y ahí es donde entran los principios de diseños que mencionábamos con anterioridad.

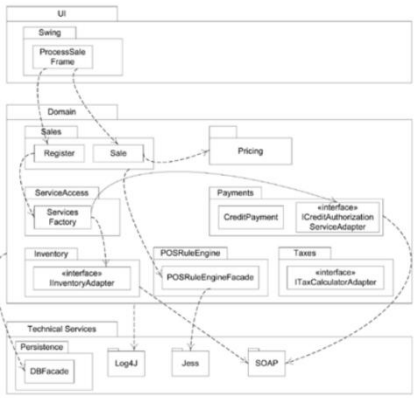
De los requisitos al diseño

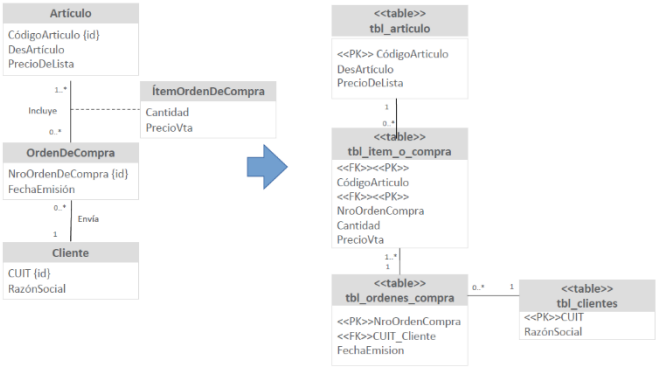
Un enfoque posible

1. Realizar un diseño arquitectónico inicial, identificando los grandes subsistemas, base de datos, clases significativas, etc. que tienen que ver con el dominio del problema que estamos analizando.
2. Realizar el diseño de la base de datos.
3. Realizar el diseño de las colaboraciones necesarias para resolver/implementar cada escenario (casos de uso, user stories, features, etc.), y para eso lo que vamos a tener que identificar son los principales elementos, algunos de esos elementos que van a participar para resolver la funcionalidad que nos pide cada escenario/historia de usuario (clases, subsistemas).
4. Realizar el diseño de detalle de cada uno de los elementos (atributos, métodos, interfaces, etc.).
5. Refinar la arquitectura.
6. ¡Programar, probar y refinar el diseño!

1. Realizar un diseño arquitectónico inicial

Nosotros deberíamos poder definir esto, poder definir cuáles son los grandes subsistemas que tenemos. Por supuesto que si vamos por un enfoque tipo Model View Controller y a través de alguna herramienta o algún entorno, bueno, probablemente esto ya lo tenemos más o menos resuelto.





2. Realizar el diseño de la base de datos

Después hay que poder determinar, transformar ese modelo conceptual que tenemos en el modelo de dominio, en un modelo lógico de la base de datos

3. Diseñar las colaboraciones

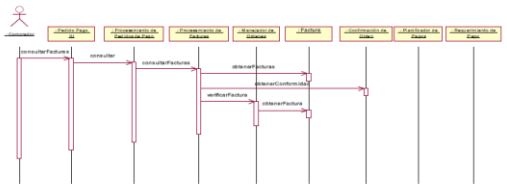
Después, obviamente para cada una de las historias de usuario, para cada uno de los escenarios que tengamos, poder transformar eso en… bueno, a ver, cuáles son los objetos que necesitamos, cuáles son las clases que necesitamos, qué subsistemas externos participan en la implementación de este escenario.

Como <actor>, quiero poder <función> de modo que/para así <razón>

Dado que <contexto>, cuando <acción> entonces <consecuencias>

Dado que <contexto>, cuando <acción> entonces <consecuencias>

Dado que <contexto>, cuando <acción> entonces <consecuencias>



Y por supuesto que esto es iterativo, es incremental y hay que paulatinamente ir refinando. Así que con este vistazo muy general, lo que pretendemos es que se lleven una idea y puedan profundizar a través de las lecturas que implica transformar el requisito en diseño. El diseño, en definitiva, es una definición ya cercana a la implementación de lo que tiene que hacer el software y de cómo lo va a ser. Fundamentalmente de acá hay más bien un “cómo”. El “qué”, si se quiere, está en los requisitos.

Conclusiones

Resumen

- En la práctica, la ingeniería de requisitos y el diseño ocurren simultáneamente y se realimentan entre sí.
- El resultado del diseño debe ser un diseño.
- El código es, en cierta medida, un tipo de modelo particular que refleja las decisiones de diseño.
- El nivel de formalidad del diseño dependerá de múltiples factores: complejidad técnica, riesgo, costos, tamaño y experiencia del equipo, etc.

[**V** / F] El diseño de software es un proceso junto a los resultados de un trabajo iterativo e incremental