

Entendiendo el problema: Principios de Diseño

Los clientes o usuarios no pueden hacer el diseño por nosotros, pero si pueden mostrarnos los problemas que tienen. Debemos entender bien el problema y poder entonces proponer una solución.

Para lograr esto, tenemos que:

- Saber bien quién es nuestro cliente o usuario. Así, no hacemos suposiciones y podemos entregar exactamente lo que el usuario quiere
- Detectar los Puntos de Dolor del usuario.
- Establecer objetivos.
- Tratar de definir alguna métrica que indique que se resuelve dicho punto de dolor o cumple con el objetivo.

Es muy importante también validar nuestras suposiciones para entender bien el problema y ver si nuestra solución realmente lo resuelve.

Al descubrimiento del problema se le llama **Product Discovery**: construir el producto correcto

Sus riesgos son:

- Que lo que hagamos no sea valioso, no resuelve un problema relevante.
- Que la gente no sepa utilizar el producto.
- Que no contemos con el tiempo, habilidades o tecnología para hacerlo.
- Que no sea rentable o sostenible

Las prácticas y técnicas para lograr el Product Discovery son:

- Entrevistas a clientes.
- Customer Journeys.
- Mapa de Empatía.

Design Thinking

Para generar ideas innovadoras, se utiliza el método de Design Thinking, que consta de dos partes: encontrar el problema y pensar soluciones.

Este método es multidisciplinar, centrado en el usuario y se trata de innovación.

Las 5 instancias del Design Thinking:

Parte 1: Encontrar el problema

1. Empatizar: Se explora el contexto, se comprenden las necesidades realizando una investigación cual/cuantitativa. Los métodos incluyen entrevistas, mapa de empatía, shadowing, etc.
2. Definir: Se detectan oportunidades como patrones de conducta, insights, etc. Se utiliza el método de Point of View.

Breakpoint – Parte 2: Pensar soluciones

3. Idear: Proponer soluciones según el contexto y las necesidades. No se busca la idea correcta, sino que crear la mayor de ideas posibles. Método brainstorming.
4. Prototipar: Seleccionar la idea con la mejor propuesta de valor y convertirla en prototipo
5. Testear: Hacer parte al usuario, co-crear en base a la prueba y error.

Método Point of View

La declaración debe enmarcar un problema con un enfoque directo, ser inspiradora para trabajar en la solución y describir que piensa y que hace el usuario. La idea sería la siguiente:

El usuario

necesita

problema

porque insight.

Ejemplos:

Martín es un ejecutivo que vive en Bariloche, tiene dos perros, Bruno y Luna (usuario). Por su trabajo, viaja al menos dos veces por mes a Buenos Aires. Martín necesita encontrar al cuidador ideal (necesita) que pueda mantener la rutina de sus perros, (problema) porque él piensa mucho en ellos y se siente mal cuando los deja con desconocidos mientras está de viaje (insight).

Juan (usuario) necesita tener una tarjeta de débito no bancaria (necesita) porque no pertenece al sistema financiero y se tiene que manejar en efectivo(problema) de esta forma podrá hacer compras por internet que viene postergando (insight).

Método Brainstorming

Esta técnica se usa cuando se necesita generar un gran número de ideas, liberar la creatividad el equipo, involucrar oportunidades de mejora, discutir conceptos nuevos, etc.

Tiene 4 reglas fundamentales

- Toda crítica está prohibida
- Toda idea es bienvenida
- Pensar en tantas ideas como sea posible
- El desarrollo y asociación de ideas es deseable => resonancia selectiva.

Buenas Prácticas de Código

Ejemplo: analizar el siguiente extracto de código e identificar puntos de mejora

```
namespace Model.Movies
{
    /// <summary>
    /// Summary description for Customer.
    /// </summary>
    public class Customer
    {
        /// <summary>
        /// Name of the Rental.
        /// </summary>
        private string _n;

        /// <summary>
        /// Getter of the Rental.
        /// </summary>
        public string getName()
        {
            return this._n;
        }

        private ArrayList _rentals = new ArrayList();
        /// <summary>
        /// Agrega una renta a la lista de rentas
        /// </summary>
        public void addRental(Rental rental)
        {
            this._rentals.Add(rental);
        }
        /// <summary>
        /// Constructor
        /// </summary>
        public Customer(string name)
        {
            this._n = name;
        }

        /// <summary>
        /// Genera el reporte
        /// </summary>
        public string GetReport()
        {
            double sum = 0; int cant = 0;
            String result = "Rental Record for " + this.getName() + "\n";
            foreach(Rental rental in this._rentals)
            {
                double thisAmntAcumm = 0;
                //determine amounts for each line
                switch (rental.Movie.PriceCode)
                {
                    case Movie.REGULAR:
                        thisAmntAcumm += 2;
                        if (rental.DaysRented > 2)
                            thisAmntAcumm += (rental.DaysRented - 2) * 1.5;
                        // if (rental.DaysRented > 4)
                        // thisAmntAcumm += (rental.DaysRented - 3) * 2.5 - 2;
                        break;
                    case Movie.NEW_RELEASE:
                        thisAmntAcumm += rental.DaysRented * 3;
                        break;
                    case Movie.CHILDRENS:
                        thisAmntAcumm += 1.5;
                        if (rental.DaysRented > 3)
                            thisAmntAcumm += (rental.DaysRented - 3) * 1.5;
                }
            }
        }
    }
}
```

```

        break;
    }
    // add frequent renter points
    cant++;
    // add bonus for a two day new release rental
    if ((rental.Movie.PriceCode == Movie.NEW_RELEASE) &&
        rental.DaysRented > 1) cant++;
    //show figures for this rental
    result += "\t" + rental.Movie.Title + "\t" +
              thisAmntAcumm.ToString() + "\n";
    sum += thisAmntAcumm;
}

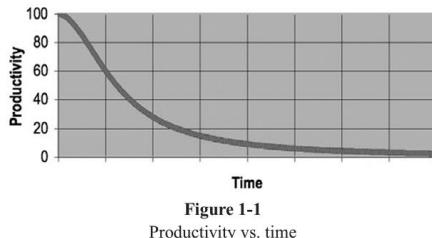
//add footer lines
result += "Amount owed is " + sum + "\n"; result += "You earned " + cant +
" frequent renter points";
return result;
}
}

```

- Nombres de variables poco representativos
- Los break adentro del for each
- Las llaves sin indentación
- Poca modularización de la función GetReport
- Mezcla de código en inglés y en español
- Las responsabilidades de la clase Customer (por ej generar un reporte) no debería corresponderle
- Estamos trabajando con objetos y se usa un switch, por lo tanto, no se está aplicando polimorfismo

No basta con solo escribir código correctamente, sino que siempre hay que ir mejorándolo.

En general el escribir mal código surge del apuro y de querer entregar un producto a las apuradas. Si no se resuelve el problema de raíz, cuando se empiezan a extender las funcionalidades del programa se vuelve más y más difícil arreglar los bugs. Puede llevar incluso a que nuestro producto se discontinue.



Querer ahorrarse mucho tiempo al comienzo puede resultar productivo, pero a la larga esa productividad disminuye considerablemente.

¿Qué es un buen código?

Un buen código se centra en la eficiencia, con un completo manejo de errores. Un buen código hace bien una sola cosa; un mal código intenta hacer demasiado, lo que lleva a desorden y ambigüedad.

El código tiene que ser simple y directo.

Objetivos que uno busca

- Mantenibilidad
- Simplicidad
- Claridad

- Flexibilidad
- Legibilidad

a. Buenas Prácticas

- Preferir nombres claros a que tener que poner comentarios comentarios, y nombres que sean pronunciables

```
const d = 3; // Tiempo transcurrido en días
const tiempoTranscurridoEnDias = 3;

const yyyyymmddstr = moment().format('YYYY/MM/DD');
const fechaActual = moment().format('YYYY/MM/DD');
```

- Usar nombres que revelen la intención de lo que hace esa variable/función

```
function obtenerCeldas() {
    let lista1 = new Array();
    for (let i=0; i<laLista.length; i++)
        if (laLista[i][0] == 4)
            lista1.add(laLista[i]);

    return lista1;
}
```

- Usar Searchable Names: no dejar valores fijos, usar constantes.
- Notaciones: en lenguajes viejos a veces usaban prefijos para indicar que algunos atributos eran privados o públicos, o un n para indicar que era un numero por ejemplo. Eso eran prácticas viejas para lenguajes no tipados por ejemplos, pero ya eso no tiene sentido con los IDEs.
- Nombres de clases, métodos: elegir una palabra/nombre por concepto.

fetch, retrieve, get

```
conseguirInfoUsuario();
conseguirDataDelCliente();
conseguirRecordDelCliente();
```

Funciones:

- Funciones/métodos pequeños: de pocas líneas, así son más fáciles de entender
- Funciones que hagan una sola cosa: Single Responsibility Principle. Esto también ayuda a la hora de testear
- Un solo nivel de abstracción por función: identificar distintos niveles de abstracción (es clave para reducir el tamaño de funciones y hacer una sola cosa)
- Leer de arriba hacia abajo: queremos leer el código como un cuento, que cada función sea seguida de aquellas que están en el siguiente nivel de abstracción, e ir bajando en nivel de abstracción a medida que bajamos en la lista de funciones.
- Evitar el switch; rompe la regla de una única responsabilidad SRP y el OCP. Es difícil hacer un switch cortito; incluso un switch con solo dos casos es largo para un bloque de función

Por naturaleza, los switch statements hacen N cosas. No siempre los podemos evitar, pero los podemos usar en clases de bajo nivel y no repetirlos. Esto lo logramos con **polimorfismo**

- Argumentos: cuantos menos, mejor
- Flags: En vez de tener flags, y tener que pasar booleanos como argumentos, es mejor usar **polimorfismo**, o crear funciones nuevas.

```

// FLAGS:
show(true)

//POLIMORFISMO:
showFinishedItems()
showDraftItems()

//FLAGS:
function createFile(name, temp) {
    if (temp) {
        fs.create(`./temp/${name}`);
    } else {
        fs.create(name);
    }
}

//POLIMORFISMO
function createFile(name) {
    fs.create(name);
}

function createTempFile(name) {
    createFile(`./temp/${name}`);
}

```

En vez de tener una única función que sea crear y tener un booleano como parámetro, directamente armamos una función para cada tipo, aplicando polimorfismo.

Principios a tener en cuenta

1. **DRY - don't repeat yourself:**
2. **The Principle of Least Surprise:**

Day day = DayDate.StringToDate(String dayName);

3. **The Boy Scout rule:** ir mejorando las cosas a medida que uno hace refactoring.

Mostrar(true);

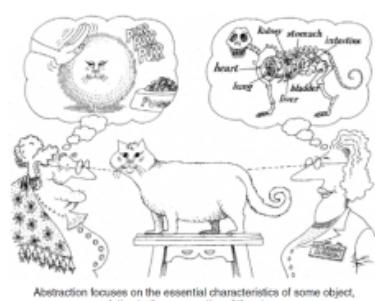
NO es suficiente escribir el código bien, el código tiene que **mantenerse limpio en el tiempo**, para que no se degrade.

Si vamos dejando nuestro código un poquito más limpio de como lo encontramos, el código no se “pudre”.

4. Abstracción:

La abstracción es una de las formas fundamentales en que nosotros, como humanos, enfrentamos la **complejidad**. Surge del reconocimiento de **similitudes** entre ciertos objetos del mundo real, y la decisión de concentrarse en estas similitudes e ignorar por el momento las **diferencias**.

Una buena abstracción **enfatiza** los **detalles que son significativos** para el usuario y **suprime** los detalles que son, al menos por el momento, **irrelevantes**.



2.1. Principios de la Abstracción

Una abstracción se enfoca en la vista exterior de un objeto y sirve para separar su comportamiento esencial de su implementación.

- **Principio de mínimo compromiso (Principle of least commitment):** la interfaz de un objeto proporciona su comportamiento esencial, y nada más.
- **Principio del menor asombro (Principle of least astonishment):** una abstracción captura todo el comportamiento de algún objeto, ni más ni menos, y no ofrece sorpresas ni efectos secundarios que vayan más allá del alcance de la abstracción.

b. Comentarios

El Uso de comentarios debe ser apropiado y conciso (justos y necesarios). No hay una regla sobre si son buenos o malos; depende del uso que les demos. Por ejemplo, son útiles para explicar funciones, pero si una función es muy explícita sobre su comportamiento, pueden estar de más.

Son buenos cuando son informativos, cuando explican una intención o sirven para clarificar/advertir, o remarcar que quedan cosas por hacer. Deben ser concisos, y que no expliquen cosas que, por ejemplo, pueden ser explicadas colocándole el nombre correcto a la variable/función.

Una vez que escribimos comentarios, ya son parte del código hay que mantenerlos. Si se modifica el código el comentario tmb habría que cambiarlo

- Informativos:
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern patronTiempo = Pattern.compile("\\\d*:\\\\d*:\\\\d* \\\\w*, \\\\w* \\\\d*, \\\\d*");
- Explicación de una intención:
// Este es nuestro mejor intento de obtener una // condición de un gran número de hilos.
- Clarificación:
assertTrue(a.compareTo(a) == 0); // a == a
assertTrue(a.compareTo(b) != 0); // a != b
- Advertencias de consecuencias:
// No correr este test a menos que
// tengas bastante tiempo (Demora).
- TODO: // TODO: Tarea a realizar

Malos comentarios

No dejar código comentado (código muerto): tener repositorios y commits atómicos; se puede volver atrás. Si uno comenta en general es porque está desarrollando, pero no hay que dejar código muerto a la hora de commitear.

- Redundancia:
// Declaro una variable "x"
let x;
- // Le sumo 1
x = x + 1;
- Comentario erróneo:
Puede introducir errores
- Comentarios obligatorios:
Tienden a que se utilicen de manera inadecuada

c. Formato

```
package fitnessse.wikitext.widgets; import
java.util.regex.*; public class BoldWidget
extends ParentWidget { public static final
String REGEXP = "'.+?'"; private static
final Pattern pattern =
Pattern.compile("'''(.+?)'''",
Pattern.MULTILINE + Pattern.DOTALL); public
BoldWidget(ParentWidget parent, String text)
throws Exception { super(parent); Matcher match
= pattern.matcher(text); match.find();
addChildWidgets(match.group(1)); } public String
render() throws Exception { StringBuffer html =
new StringBuffer("<b>");
html.append(childHtml()).append("</b>"); return
html.toString(); }
```

Esto es ilegible. Es importante ponerse de acuerdo con el equipo, que quede todo estandarizado y lindo.

A veces se puede querer que no tenga formato (como los lenguajes compilables y descompilables – ej Javascript), pero el código debe estar claro

d. Excepciones

Usar excepciones en vez de códigos de error (muy común en POO)

```
If (deletePage(page) == E_OK) { .... }
```

En general no retornan Null.

Las dos grandes ramas de excepciones son **salvables** (más las que tienen que ver con la lógica de negocio – ej debe ingresar un nro e ingresa una letra; le suelen llegar al usuario como un msj en pantalla) y **no salvables** (son errores de aplicación en general; el programa debe ser terminado y no se puede recuperar – ej el archivo no existe, la bdd está caída, etc.)

```
public void delete(Page page) {
    try {
        deletePageAndAllReferences(page);
    }
    catch (Exception e) {
        logError(e);
    }
}

private void deletePageAndAllReferences(Page page) throws Exception {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
    logger.log(e.getMessage());
}
```

In the above, the `delete` function is all about error processing. It is easy to understand and then ignore. The `deletePageAndAllReferences` function is all about the processes of fully deleting a `page`. Error handling can be ignored. This provides a nice separation that makes the code easier to understand and modify.

e. Test unitarios

F.I.R.S.T.

Fast
Independent
Repeatable
Self-Validating
Timely

Ejemplos:

WhatAreWeTesting_InWhatConditions_WhatAreExpectedResult

Login_ExistingUsernameWithIncorrectPassword_ShouldReturnMessageWrongPassword

- Siempre escribe casos de prueba aislados
- Prueba una sola cosa en un solo caso de prueba
- Utiliza un único método de assert por caso de prueba
- Utiliza una convención de nombres para los casos de prueba
- Utiliza mensajes descriptivos en los métodos de assert
- Mide la cobertura de código para encontrar casos de prueba faltantes

Criterios de Buen Diseño

Es importante tener un buen diseño para tener un delivery rápido, para manejar el cambio y para lidar con la complejidad.

Síntomas de un Software con mal diseño

- **Rigidez:** Yo cambio una parte del sistema y tengo que cambiar en cascada en varias otras partes; no está bien encapsulado
- **Fragilidad:** cuando uno hace un cambio en un lugar y se rompe algo en otra parte; suele pasar cuando hay mala separación de responsabilidades, código duplicado, etc. Este es peor porque ni siquiera te das cuenta de que las porciones de código están acopladas
- **Inmobilidad:** No poder llevar clases/métodos a otros sistemas por no darse cuenta que se tienen cosas acopladas, lo que fuerza a que te tengas que llevar no solo la responsabilidad que queres sino todas sus acopladas también
- **Viscosidad:** la viscosidad indica que el sistema está volviéndose rígido o difícil de mantener, lo que lleva a decisiones subóptimas que pueden incrementar la deuda técnica y disminuir la calidad del código. Puede ser viscosidad del diseño o del ambiente.

La viscosidad del diseño es alta cuando, si hay que elegir soluciones para hacer un cambio en el código, la preservación de métodos es más difícil de hacer que los cambios.

La viscosidad del ambiente es cuando el ambiente de desarrollo es lento e inefficiente (si los tiempos de compilación muy largos, los desarrolladores pueden querer hacer cambios para acortar esos tiempos, por mas que esos cambios no sean óptimos desde un POV de diseño)

En general que un sistema se vuelva rígido, frágil e inmóvil tiene que ver con la incorrecta dependencia entre módulos

Cómo lograr un buen diseño: SOLID



- **SRP: Single Responsibility Principle**

Que se haga una sola cosa

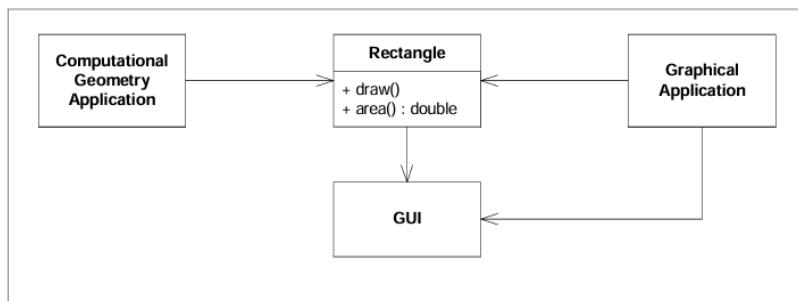


Figure 9-1
More than one responsibility

Este diseño viola SRP porque la clase Rectangle tiene dos responsabilidades: dibujar el rectángulo y calcular el área.

Es mejor separar cada responsabilidad en clases distintas.

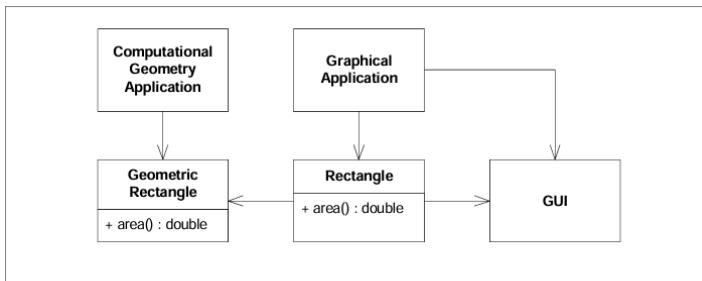


Figure 9-2
Separated Responsibilities

Ejemplo 2:

```

public class OrderController
{
    ...
    public ActionResult CreateForm()
    {
        return View();
    }

    [HttpPost]
    public ActionResult Create(OrderCreateRequest request)
    {
        if (!ModelState.IsValid)
        {
            /* View data preparations */
            return View();
        }
        using (var context = new DataContext())
        {
            var order = new Order();
            // Create order from request
            context.Orders.Add(order);

            // Reserve ordered goods
            ... (Huge logic here)...

            context.SaveChanges();

            //Send email with order details for customer
            var smtp = new SMTP();
            // Setting smtp.Host, UserName, Password and other parameters
            smtp.Send(client, order);
        }
        return RedirectToAction("Index");
    }
    ... (many more methods like Create here)
}

```

Vs:

```

public class OrderService
{
    public void Create(OrderCreateRequest request)
    {
        // all actions for order creating here
        using (var context = new DataContext())
        {
            var order = new Order();
            // Create order from request
            context.Orders.Add(order);
            // Reserve ordered goods
            ... (Huge logic here)...
            context.SaveChanges();

            //Send email with order details for customer
            var smtp = new SMTP();
            // Setting smtp.Host, UserName, Password and other parameters
            smtp.Send();
        }
    }
}

public class OrderController
{
    public OrderController()
    {
        this.service = new OrderService();
    }
    [HttpPost]
    public ActionResult Create(OrderCreateRequest request)
    {
        if (!ModelState.IsValid)
        {
            return View();
        }
        this.service.Create(request);
        return RedirectToAction("Index");
    }
}

```

Enviar un correo electrónico, en realidad, no es una parte del flujo de creación del pedido principal. Incluso si la aplicación no logra enviar el correo electrónico, la orden sigue creándose correctamente.

Además, podría surgir una nueva opción en el área de configuración de usuario que les permite optar por no recibir un correo electrónico después de realizar un pedido con éxito, o incluso otros medios.

Puede pensarse en un Helper para delegar el envío, o para desacoplar más un, un Observer o Eventos.

Responsabilidad = una razón de cambio.

Si pensamos que puede haber mas de un motivo por el cual cambiar una clase, entonces esa clase tiene mas de una responsabilidad.

- **OCP: Open Closed Principle**

Una clase debe estar abierta a la extensión y cerrada a la modificación. En el mundo ideal: si el código que escribí anda, si yo quiero agregar nuevas funcionalidades puedo, pero sin tocar/romper el código ya escrito.

- ✓ Abierto a la extensión: el comportamiento del modulo se puede extender; lo podemos hacer comportarse de nuevas y diferentes maneras a medida que cambian las necesidades de la aplicación.
- ✓ Cerrado a la modificación: el código fuente no se cambia; no se puede cambiar nada del modulo

Típica violación de OCP:

```
public double Area(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape; area +=
rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape; area += circle.Radius * circle.Radius *
Math.PI;
        }
    }
    return area;
}
```

Si quisiera agregar una nueva figura tengo que cambiar el código que ya tengo para extender las funcionalidades.

Usa el pilar de abstracciones para aplicar polimorfismo. En general si vemos un switch podemos intuir que se está violando este principio (porque si quiero agregar algo tengo que agregar un caso más y eso es tocar el código viejo ya cerrado que funcionaba)

```

public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width*Height;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius*Radius*Math.PI;
    }
}

public double Area(Shape[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        area += shape.Area();
    }
    return area;
}

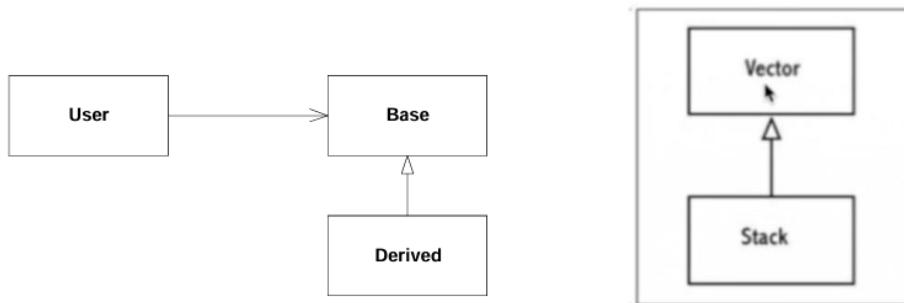
```

Ahora también puede ser que yo quiera aplicar **KIS (keep it simple)**, en ese caso capaz no aplico tanto Open-Close, depende lo que yo quiera cubrir más, y cuales sean las necesidades del cliente qué principio voy a priorizar (**YAGNI: You ain't gonna need it**)

- **LSP (Liskov Substitution Principle)**

las subclases deben poder ser sustituidas por las clases base.

Cuando podés cambiar una clase por otra y esperar que el comportamiento sea el mismo. Se basa, si pensamos en el diagrama de herencia, a un “is a”.



A stack is a vector.

Si el usuario toma un argumento de tipo **Base**, debería ser legal pasarle una instancia de **derived** a esa función

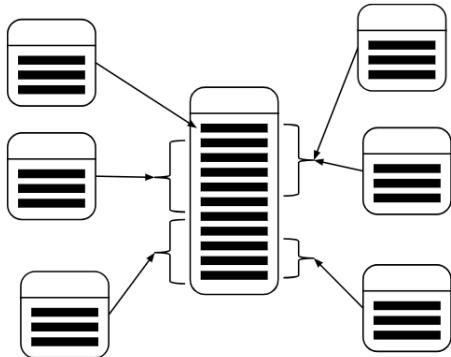
Si hay algo de Vector que no aplica para Stack, hay algo que no estoy cumpliendo de la herencia, y por lo tanto no cumple LSP.

Si uno usa herencia, se debe cumplir realmente el *is a* y todo eso; no puedo tener una parte si y una parte no

- **ISP: Interface Segregation Principle**

Las interfaces deben estar bien modularizadas para que uno no tenga que implementar métodos que no va a utilizar.

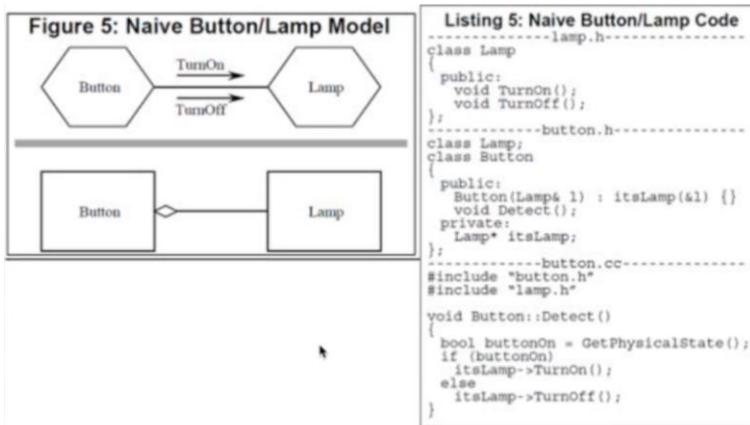
Un síntoma de que se está violando este principio es, por ejemplo, si tengo una interfaz muy grande y tengo muchas clases que se conectan a ella, pero solo usan un pedacito. Es mejor que cada clase tenga su interfaz con lo que va a necesitar de ella.



- **DIP: Dependency Inversion Principle**

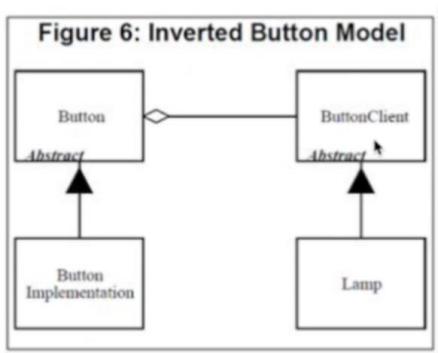
Algo de alto nivel no debería depender de un módulo de bajo nivel.

Ejemplo: Botón que controla una lámpara



El problema de esto es que este botón solo puede prender y apagar lámparas. Si yo lo quisiera reutilizar para prender/apagar una tele no puedo, porque el código solamente está hecho para ese caso particular. El botón, que es algo más genérico/abstracto depende de la lámpara y debería ser al revés.

Lo ideal es desacoplarlo:



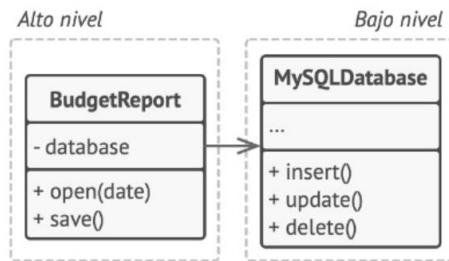
Ahora la lámpara depende del botón, a través de una abstracción (invertimos la dependencia; ahora la lámpara es la que tiene el botón, de alguna manera). Ahora puedo usar una tele que implemente encendible, y un botón puede prender/apagar cualquier encendible.

- Una vez que las clases de bajo nivel implementan esas interfaces, se vuelven dependientes del nivel de la lógica de negocio, invirtiendo la dirección de la dependencia original.

El principio de inversión de la dependencia suele ir de la mano del *principio de abierto/cerrado*: puedes extender clases de bajo nivel para utilizarlas con distintas clases de lógica de negocio sin descomponer clases existentes.

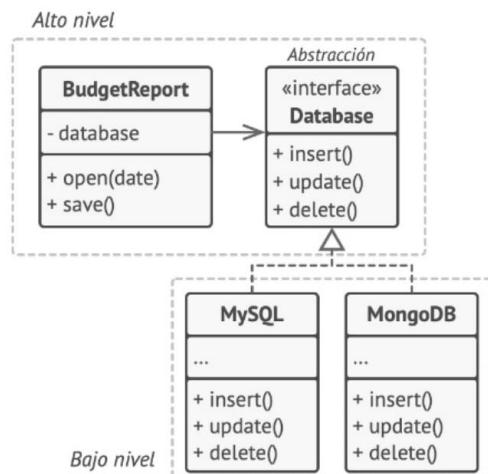
Ejemplo

En este ejemplo, la clase de alto nivel – que se ocupa de informes presupuestarios – utiliza una clase de base de datos de bajo nivel para leer y almacenar su información. Esto significa que cualquier cambio en la clase de bajo nivel, como en el caso del lanzamiento de una nueva versión del servidor de la base de datos, puede afectar a la clase de alto nivel, que no tiene por qué conocer los detalles de almacenamiento de datos.



ANTES: una clase de alto nivel depende de una clase de bajo nivel.

Puedes arreglar este problema creando una interfaz de alto nivel que describa operaciones de leer/escribir y haciendo que la clase de informes utilice esa interfaz en lugar de la clase de bajo nivel. Despues puedes cambiar o extender la clase de bajo nivel original para implementar la nueva interfaz de leer/escribir declarada por la lógica de negocio.



DESPUÉS: las clases de bajo nivel dependen de una abstracción de alto nivel.

Como resultado, la dirección de la dependencia original se ha invertido: las clases de bajo nivel dependen ahora de abstracciones de alto nivel.

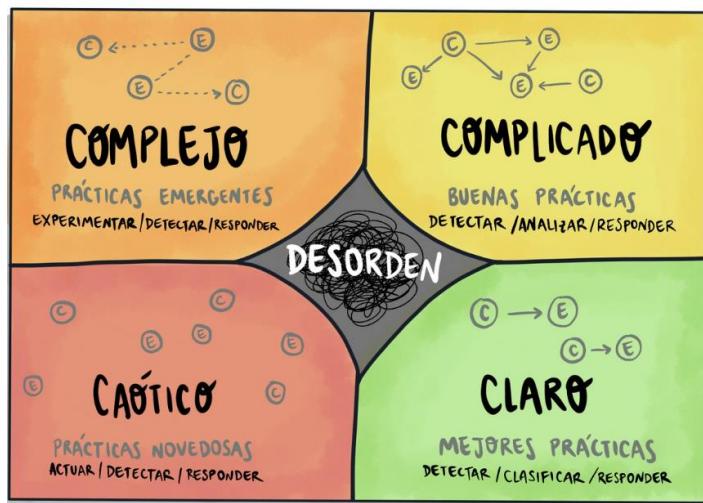
Conclusiones

- Un buen diseño es necesario para lidiar exitosamente con los cambios
- Las principales fuerzas que conducen el diseño deberían ser ALTA COHESIÓN y BAJO ACOPLAMIENTO (en un contexto dado para un problema dado)
- Los principios SOLID nos ponen en el camino correcto

Agilidad y Metodologías SCRUM

Agilidad

Marco Cynefin



Es una manera de separar cualquier problema/situación.

- Claro: aplicas la receta de algo que ya conoces, ya entendes como resolverlo, seguis los pasos y lo resolves
- Complicado: podes llegar al resultado deseado, analizando, estudiando qué es lo que hay que hacer para resolverlo. Ejemplo: necesito cruzar de un lado a otro y construyo un puente

De la mitad para la izquierda son cosas que por más que yo planifique es muy difícil que lo logre. En lo complejo o lo caótico, hay cosas que no dependen del plan de las personas. Acá ya hay mas ambigüedades en los requerimientos, hay una problemática con los recursos (ejemplo se rompe el servidor)

Agilidad

- NO es una metodología, porque no es algo que pretenda indicarnos qué hacer paso a paso. No hay un único paso a paso que nos sirva, hay muchas soluciones.
- Va más allá de los marcos de trabajo (scrums, dailys, etc)
- No es ir más rápido, aunque **entregar valor temprano** sí es una de las promesas ágiles
- No es multitasking, aunque implique gestionar un contexto complejo, incierto y cambiante.

La agilidad es una **forma de hacer** que nos permite experimentar y aprender en pasos pequeños y una **forma de ser** en la que nos ayudamos y mejoramos en nuestro entorno. Es un **mindset** de enfocarse en dar valor, trabajando colaborativamente. También son valores y principios.

Desarrollo de productos: tradicional vs ágil

La tradicional tiene etapas muy específicas:

- Análisis de alto nivel
- Análisis detallado
- Diseño
- Construcción
- Pruebas
- Implementación

Por ejemplo, esto puede llevar un año. Esto funciona perfecto para ámbitos donde no hay cambios de prioridades, por ejemplo. Acá tenemos un alcance y hay que hacerlo todo

En la ágil presentamos varios cambios:

- Priorizar
- Inceptions: kick-off para entendimiento del problema pero un poco más detallado y participativo para que todos los del equipo comprendan el problema
- Se va trabajando en iteraciones/ciclos de trabajo, y se va entregando a lo largo del proyecto
- Así, podemos ir agregando valor en las iteraciones



Si se corta el proyecto a la mitad, en la 1ra quedaste en desarrollo 0; en la ágil fuiste entregando según tu prioridad

Extreme Programming XP

Es una metodología de desarrollo de software que enfatiza la colaboración, simplicidad y flexibilidad. Se focaliza en entregar pequeñas versiones de un producto de software en ciclos cortos. Incluye técnicas de pair programming y refactorización de código.



Copyright 2000 J. Donvan Wells

Kanban

Un tablero que ayuda a visibilizar el trabajo dentro de un equipo, para poder trabajar en sintonía. Puede ir dando lugar a que tengamos un proceso más claro, con políticas definidas y con asignación de tareas para determinados integrantes del equipo, con un seguimiento.

Tiene tres columnas: To Do, WIP, y Done

Scrum

Hay distintas variantes de implementar metodologías ágiles. Pueden ser por ejemplo SCRUM, Kanban, etc. Lo que hacen las metodologías agiles es **proveer herramientas para poder resolver problemas que a priori parecen complejos de forma iterativa e incremental**. Buscan resolver el problema de a pedacitos.

Permiten que el producto adquiera una funcionalidad mínima relativamente rápido, permitiendo la detección de errores prematura, manteniendo la comunicación continua con el cliente, etc. Así también surge el MVP (Minimum Viable Product), y luego ir agregando features.

Es un framework de metodologías ágiles. Propone un marco de trabajo (conjunto de reglas para trabajar en un ámbito controlado) y se centra en la idea de que “todo el equipo tira para el mismo lado”.

Modelo Cynefin:



- Simple: conjunto de pasos bien definidos
- Complicado: se analiza el problema y vienen personas expertas en el problema para ejecutar esos pasos más complejos.
- Complejo: no puedo determinar causa-efecto. El efecto yo lo veo una vez que ocurrió cierto evento. Requiere de la prueba-error para conocer el resultado
- Caótico: estoy en una situación de emergencia. Ya no se puede pensar en pasos. Hay que actuar.

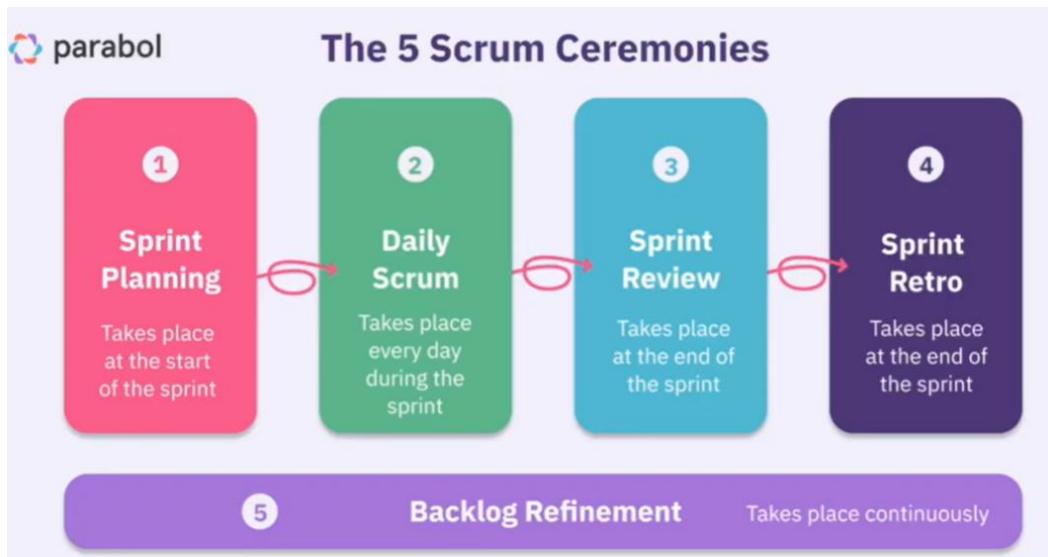
Se podría decir que las metodologías agiles funcionan bien cuando se trata de un problema complejo; cuando tengo que experimentar. Vamos diseñando, implementando, haciendo prueba y error hasta que se cumplen las expectativas del usuario.

Scrum Framework



- Product Owner: arma las historias de usuario para describir las features de lo que quiere el usuario, para así poder iniciar la búsqueda
- Producto backlog: formalización de lo que solicita el usuario. Puede estar en desorden lo que se mete en el backlog; acá las cosas pueden ser que aún no estén bien definidas.
- **Refinement:**

Ceremonias



1. **Sprint planning**: se agarran del backlog algunas historias para planear lo que se hará en el próximo sprint de desarrollo. Se van a ir definiendo y especificando historias.

Se planifica la capacidad para decidir cuánto tiempo se puede asignar al sprint. También se define el objetivo del sprint, cual es el resultado que se espera.

Sprint backlog: Las historias de usuario se agregan al sprint backlog en base a un acuerdo colectivo del equipo. Esta es una oportunidad para que el equipo haga preguntas para aclarar lo que se pide en la historia de usuario si no se cubrió completamente en la refinación del backlog.

2. **Daily Scrum**: que hizo el equipo el día anterior, como vienen etc.

Reunión diaria donde se juntan el PO, Scrum Master y el equipo de desarrollo para revisar el trabajo recientemente completado y establecer qué se va a hacer ese día.

3. **Sprint review:** Una vez finalizado un sprint, se hace una review, que suele ser una demo al usuario y al PO.

Se hace el último día del sprint; se revisa lo que se logró completar, en general a través de una demo de lo nuevo que se implementó, y se lo muestran a los stakeholders

4. **Sprint Retro:** reunión para el propio equipo (interna) para analizar cómo salió el sprint anterior

Se reflexiona sobre los logros/dificultades del sprint, para explorar oportunidades de mejora. ¿Qué salió bien? ¿Qué salió no? ¿Qué podemos mejorar?

5. **Backlog refinement:** se puede hacer tanto al principio como al final del sprint.

Esta reunión permite al Product Owner guiar al equipo en la refinación de las historias de usuario con mayor prioridad en preparación para los sprints futuros

Otro concepto: Deuda técnica - estamos desarrollando y nos topamos con un problema complicado. Lo resolvemos un poco por encima para salir del paso y poder seguir con otros features, pero no podemos olvidar que después tenemos que volver a resolverlo

En resumen:

Ceremonia	Propósito	Asistentes	Consejos y Trucos
Planificación del Sprint	Identificar los objetivos del sprint y crear el backlog del sprint.	Todo el equipo Scrum	Fomentar que los miembros del equipo discutan y negocien elementos relacionados con el objetivo del sprint.
Scrum Diario	Facilitar que el equipo Scrum discuta el progreso y anuncie compromisos diarios.	Scrum Master y equipo de desarrollo	No permitir que la reunión exceda los 15 minutos.
Revisión del Sprint	Mostrar el trabajo completado durante el sprint.	Todo el equipo Scrum con Product Manager, stakeholders y clientes	Capturar retroalimentación actionable como elementos en el backlog.
Retrospectiva del Sprint	Permitir que el equipo Scrum inspeccione y planifique mejoras para el próximo sprint.	Scrum Master y equipo de desarrollo	Asegurarse de que se capturen, asignen y hagan seguimiento a las sugerencias actionable.

*todo el equipo Scrum: PO, líder y IT

Métricas

Se van tomando métricas a lo largo de todo el desarrollo (en general se recopilan al final de cada sprint) y, por ejemplo, va recopilando los story points, para ver cuántos puntos se fueron ejecutando en cada sprint.

Así, podemos ver si de un sprint a otro se ejecutaron más o menos puntos; si de un sprint a otro puede ser porque estimaron mal, porque era compleja, etc.

Product Discovery

Distintos ciclos de vida de desarrollo nos llevan a distintos valores del producto. Si trabajamos en sprints más cortos, si en alguno fallamos, podemos corregirlo en menor tiempo y volver a intentar. Tengo que contemplar que cuando salgo con un producto ese no resuelve el problema que habíamos planteado. Así, surge el concepto de:

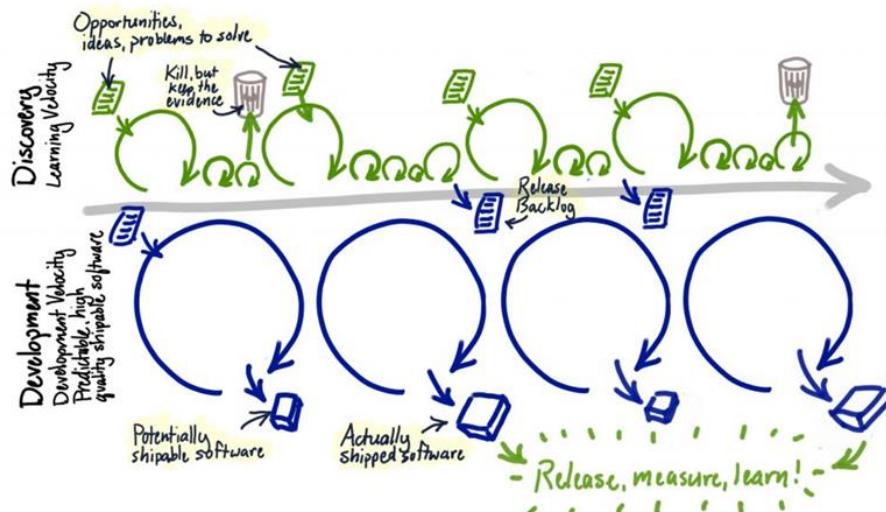
MVP (Minimum Viable Product): Ir entregando pequeñas versiones de manera iterativa, que se vayan ajustando.

Este concepto resulta medio ambiguo y a veces se reemplaza por una terminología más específica para lo que hace cada versión que se lanza, que luego de cada iteración se pueden ir clasificando como:

- Testable
- Usable
- Querido

El **Product Discovery** en muchos lugares solo se hace al inicio, para averiguar qué es lo que resolvería la necesidad del usuario, pero en la práctica debería ser [continuo en el tiempo](#).

Discovery, no solo al inicio ([Dual Track](#))



El ciclo de arriba sería del equipo de producto, que está en Discovery, viendo qué es viable y qué no, etc. Puede ser que matamos una idea inviable, pero nos guardamos las pruebas para que otros no vayan por ese camino porque no es posible.

La idea del Discovery es que después de él, al equipo de desarrollo le lleguen cosas que fueron validadas.

Mentalidad de producto Discovery

1. [Colaboración y aprendizaje en el equipo de producto](#)

- Todo el Equipo de Producto conoce y aprende del Product Discovery, participando en las decisiones sobre la funcionalidad a crear
- Evitando guerras de opiniones y coronadas
- Desarrollando y manteniendo un entendimiento compartido
- Con artefactos de visualización

2. [User centered design](#)

- El equipo de producto trabaja en comunicación directa con usuarios y clientes

- No solo validando problemas y soluciones sino además co-creando con ellos
- Diseñar experiencias

3. Humildad intelectual

- Tolerancia a la incertidumbre: no hay certezas sino hipótesis a validar
- Experimentos: tolerancia al error y a cambiar de parecer
- Construir un ambiente seguro e inclusivo de respeto a las opiniones de todo el equipo

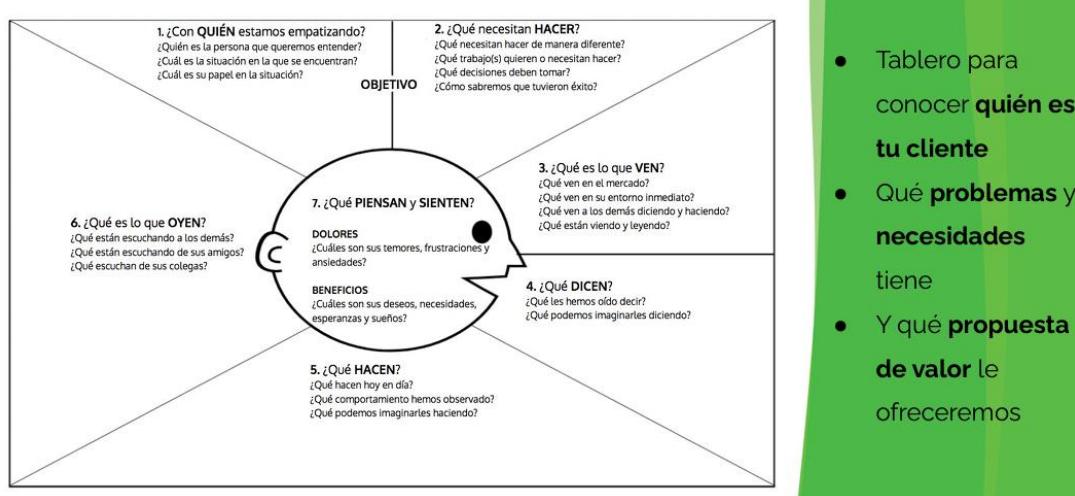
Personas (UX Personas)

Hacer un detalle del perfil de los usuarios que van a usar mi producto; recopilar información que va a ser relevante para el desarrollo de nuestro producto.

Tengo que analizar bien **quiénes son mis clientes**.

- Hacer **foco en segmentos específicos**
- Técnica para comprender las diferentes necesidades de los principales **tipos de clientes** de un producto
- También sirve para generar conversaciones valiosas al momento de definir y priorizar las oportunidades.

Mapa de Empatía



Puede ser que según cierto perfil de usuario que tenga, decida priorizar cierto desarrollo para esos perfiles, y luego para el próximo sprint seguir con las necesidades de otro tipo de perfil.



Historias de Usuario

Historia de usuario: descripción de la funcionalidad del sistema, desde el punto de vista de un usuario:

Como <persona>
quiero <función>
para <objetivo>

Nos dan una idea de que valor queremos proveerle al usuario. Se enfocan en el valor provisto, dando pie a discutir el requisito No son el requisito en sí

Ejemplos:

Como Carla (estudiante universitaria)

quiero consultar el listado de cursos disponibles de una materia

para decidir a qué curso anotarme

Como Juan Carlos (jubilado)

quiero imprimir un comprobante de mi reserva de turno

para tener recordatorio de mi turno si mis dispositivos electrónicos fallan

Framework QUS (Quality User Story)

Cuáles son los **atributos de calidad** de las historias de usuario?

Calidad Sintáctica	Bien formadas	Atómicas
	Mínimas	
Calidad Semántica	Conceptualmente Acertadas	Orientadas al Problema
	Sin Ambigüedades	Sin Conflictos
Calidad Pragmática	Usan Oraciones Completas	Estimables
	Únicas	Uniformes
	Independientes	Completas

Queremos historias orientadas al problema y no a la solución. Tampoco queremos tener historias que dupliquen a otras.

Épicas

Son historias de usuario que se pueden descomponer en más historias de usuario. Deseamos subdividir las épicas hasta llegar a tareas de tamaño razonable.

Las historias de usuarios que son mas complejas, las descomponemos en **EPICAS**: anidamos historias en historias más grandes.

Ejemplo:

- Como estudiante, quiero inscribirme en materias para el cuatrimestre para...
 - ... quiero saber cuando me corresponde inscribirme ...
 - ... quiero saber en qué cursos puedo inscribirme ...
 - ... quiero saber qué materias hay en mi plan de estudios ...
 - ... quiero saber qué materias tengo aprobadas ...
 - ... quiero saber qué cursos tienen cupo disponible para cada materia ...
 - ... quiero inscribirme en cursos ...

Se recomienda que las tareas del backlog cumplan:

I	ndependent	Separable de otras tareas
N	egotiable	No es innecesariamente rígido
V	aluable	Aporta valor al cliente/usuario
E	stimable	Possible aproximar su costo
S	mall	Desarrollable en una iteración
T	estable	Verificable (al menos en principio)

Malas historias de usuario

- **Mal formada:** ej: quiero descargar reportes manuales. ¿Quién lo quiere? ¿para qué?
- **Orientada a la solución:** Como analista, quiero poder exportar las compras a CSV para poder abrir los datos en excel para poder calcular un promedio para poder enviarlo cada día en un mail a los gerentes.
 - ✓ La historia presupone detalles de implementación
 - ✓ Hay otro problema más sutil en esta historia
- **Demasiado grande:** ej: como estudiante de FIUBA quiero recibirme para...

Lo que sí, hay ciertas situaciones en las que la necesidad del usuario realmente requiere una implementación específica:

Como Alice (personaje de ejemplos de seguridad informática)
quiero que mis mensajes se transmitan encriptados usando **para** evitar que mis adversarios puedan leer mis mensajes

3 Cs (Card, Conversation, Confirmation)

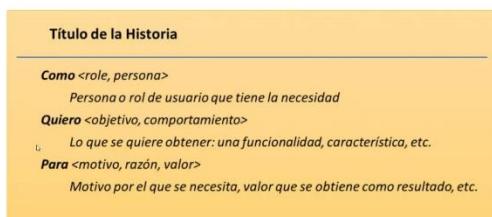
- **Card:** describe la intención del usuario.
- **Conversation:** los interesados se comunican para refinar las historias, descubriendo y documentando requisitos
- **Confirmation:** criterios de aceptación

Criterios de aceptación: condiciones específicas que deben cumplirse para aceptar el trabajo realizado. Me permiten verificar si se cumple lo que solicita el usuario.

Para documentar la lista de condiciones hay varias formas, una de ellas es plantear los **escenarios:**

- Escenarios

Dado que <contexto>
 cuando suceda <evento>
 entonces <consecuencia>



Historia Negociable 1

Como: Bloguero

Quiero: hacer una entrada al blog

Para: posicionarme como experto en un tema específico

Criterios de Aceptación:

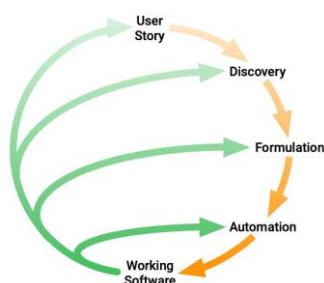
- Debo ser capaz de publicar contenido multimedia (imágenes y video)
- El texto de la entrada debe ser enriquecido (que permita enlaces Web, formato, etc.)
- La entrada se debe poder compartir vía redes sociales
- La entrada se debe poder imprimir
- La entrada se debe poder enviar vía correo electrónico

Los criterios deben ser:

- Claros
- Concisos
- Verificables
- Independientes
- Orientados al Problema
- testeable

Behaviour Driven Development

Una vez que formulamos los criterios de aceptación la idea es automatizarlo y una forma es con BDD



Ejemplo:

```

1 [D] Feature: Add money to the wallet
2 [D] Scenario: Before adding anything
3   Given an empty wallet
4   Then the wallet contains 8 dollars
5
6 [D] Scenario: Add twice
7   Given an empty wallet
8   When I add 2 dollars
9   And I add 3 dollars
10  Then the wallet contains 5 dollars
  
```

Casos de Uso

Descripción de:

- Una serie de acciones
- Realizadas por un Sistema
- Que generan un resultado observable de valor para un actor en particular

Compuestos de un escenario principal y un conjunto de escenarios alternativos.

Ejemplos:

Consultar Precio de un Producto	
El cliente obtiene información del precio de un producto	
Precondiciones	

Cliente	Sistema
1. Ingrresa el código de producto	
	2. Informa el precio del producto
Postcondiciones	

Escenarios alternativos	
2.1: Si el código de producto no corresponde a un producto registrado, el sistema informa que desconoce el producto	

O versiones más compactas:

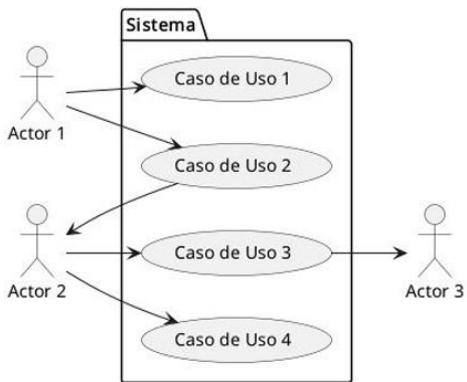
Consultar Precio de un Producto
1. El cliente ingresa el código de producto
2. El sistema informa el precio del producto
2.1. Si el código de producto no corresponde a un producto registrado, el sistema informa que desconoce el producto

Consultar Precio de un Producto
El cliente ingresa el código de producto. El sistema informa el precio del producto o que desconoce el producto

Apunta un poco a la misma idea de historias de usuarios pero va en otra dirección. Esto está más apuntado a lo que hace el sistema.

Una vez que ya decidimos de que forma queremos trabajar es una forma de documentar ese proceso.

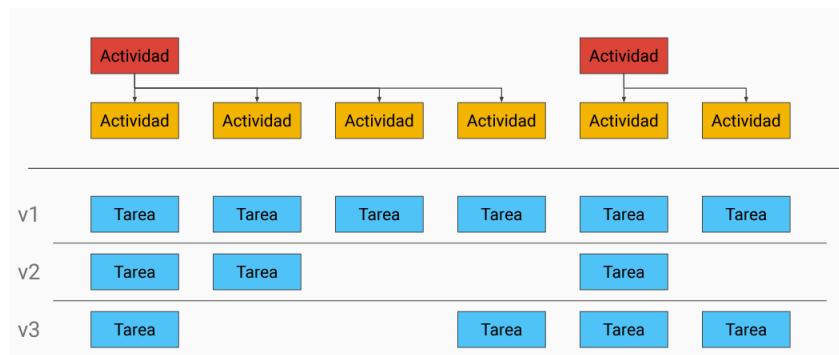
Diagramas de Casos de Uso



¿Cómo encontrar historias?

1. Establecer los límites del sistema (ej: hasta acá llega lo que quiero implementar)
2. Identificar actores involucrados en el sistema
3. Buscar objetivos y actividades
 - User story mapping
 - Impact mapping

User story Mapping

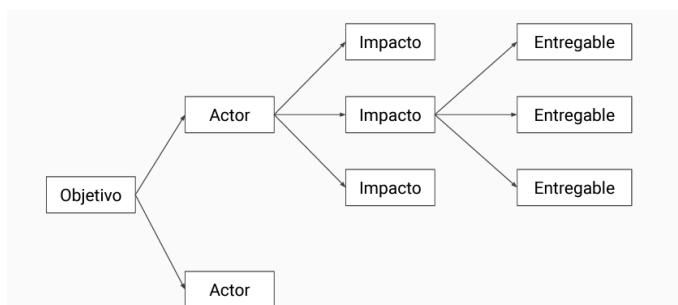


Aca tememos las tarjetas rojas y amarillas como épicas y las tareas como historias de usuario.

- Actividades: Se considera lo que un usuario va a hacer en un sistema y lo va desglosando (se registra, hace el login, etc) y luego va buscando dentro de eso diferentes actividades que eso va a hacer
- Tareas: dividimos las actividades en tareas, en lo posible pequeñas, que sean sencillas de implementar

Vas agarrando las tareas de la versión 1 para el sprint 1 y vas trabajando sobre eso.

Impact Mapping



Formas de Calcular Story Points

Los story points me dan una medida de qué es lo que el equipo va a poder lograr en el sprint, sin usar literalmente la métrica del tiempo sino midiendo otras variables también como dificultades de la tarea, tiempo, disponibilidad del equipo, todo representado por los story points.

La escala de los story points es completamente arbitraria, lo importante es que dentro de un equipo sea consistente.

POKER PLANNING:

Cada miembro del equipo necesita un conjunto de cartas de Planning Poker. Estas cartas suelen contener números de la **secuencia de Fibonacci** (1, 2, 3, 5, 8, 13, etc.) o variantes como: 0, $\frac{1}{2}$, 1, 2, 3, 5, 8, 20, 40, 100, ∞ , y una carta de interrogación.

Se selecciona un moderador (generalmente el Scrum Master) que guiará la sesión.

El equipo debe tener una comprensión básica de las historias de usuario o tareas que se van a estimar.

Revisión de la historia o tarea:

El Product Owner (PO) explica la historia de usuario o tarea a estimar. Responde preguntas y aclara dudas para asegurar que todos comprendan los requisitos.

1. Primera ronda de estimación:

Cada miembro del equipo elige en secreto una carta que representa su estimación del tamaño o esfuerzo de la tarea (por ejemplo, 3 para "mediano esfuerzo", 8 para "alto esfuerzo"). Luego, todos revelan sus cartas al mismo tiempo.

Si todos eligieron el mismo número, se registra la estimación y se pasa a la siguiente tarea. Si hay discrepancias, los miembros con las estimaciones más altas y más bajas explican su razonamiento. El equipo discute hasta alcanzar un entendimiento común sobre los factores que afectan el tamaño o esfuerzo.

2. Nueva ronda de estimación:

Después de la discusión, los miembros eligen otra carta basándose en la nueva información. Este proceso se repite hasta que el equipo alcance un consenso o una estimación aceptable para todos.

Gráficos para medir la Performance del Sprint

Las métricas se usan para visualizar e ir corrigiendo diferentes puntos que se presentaron a lo largo del sprint.

Al final de cada sprint, se miran cuantos story points se lograron y se miden las métricas

Burndown Chart

El Burndown Chart muestra el trabajo pendiente en un sprint o proyecto a lo largo del tiempo. Ayuda a visualizar si el equipo está en camino de completar las tareas planificadas.

Estructura del gráfico

- Eje X: Tiempo (por ejemplo, días del sprint o semanas del proyecto).
- Eje Y: Trabajo restante (en story points, horas, o tareas).

Línea ideal (guideline): Una línea diagonal que muestra el descenso esperado si el equipo completa el trabajo de manera constante.

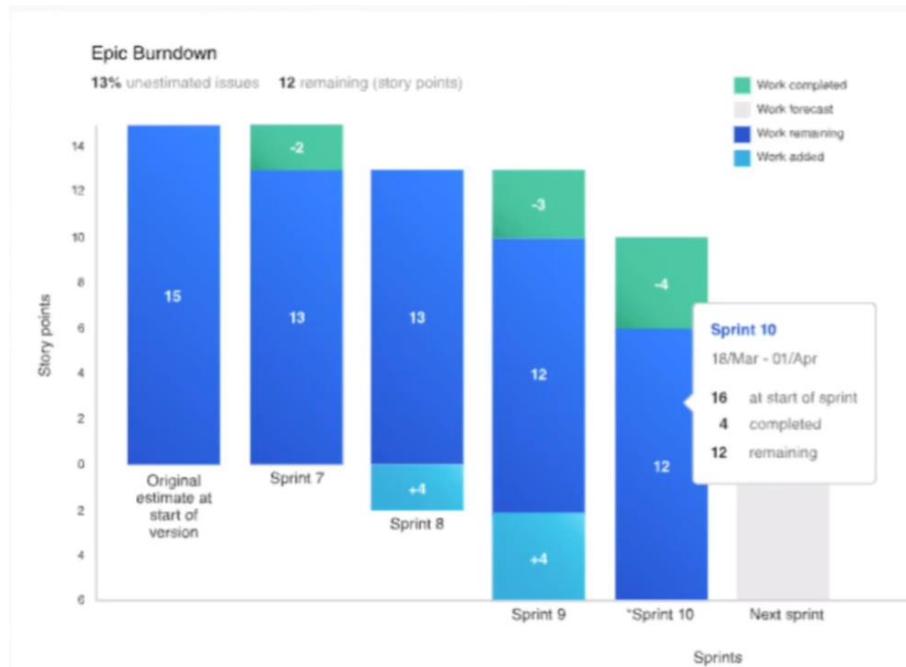
Línea real (remaining values): Refleja cómo el equipo está reduciendo el trabajo pendiente en la realidad.

Burndown Chart



- **Seguimiento diario:** El equipo actualiza el trabajo pendiente a diario, lo que refleja si están avanzando como se esperaba.
- **Detectar problemas:** Si la línea real no sigue la línea ideal (por ejemplo, se mantiene plana o baja más lentamente), podría indicar bloqueos, sobreestimación, o interrupciones.
- **Finalización del sprint:** Si la línea real llega a cero antes del final del sprint, el equipo ha completado todo el trabajo planificado.

Epic Burndown



El Epic Burndown Chart muestra cuantos puntos se fueron ejecutando en cada sprint. Esto nos puede mostrar también la velocidad en la que podemos completar story points, para ver si aumenta/disminuye la velocidad entre un sprint y otro.

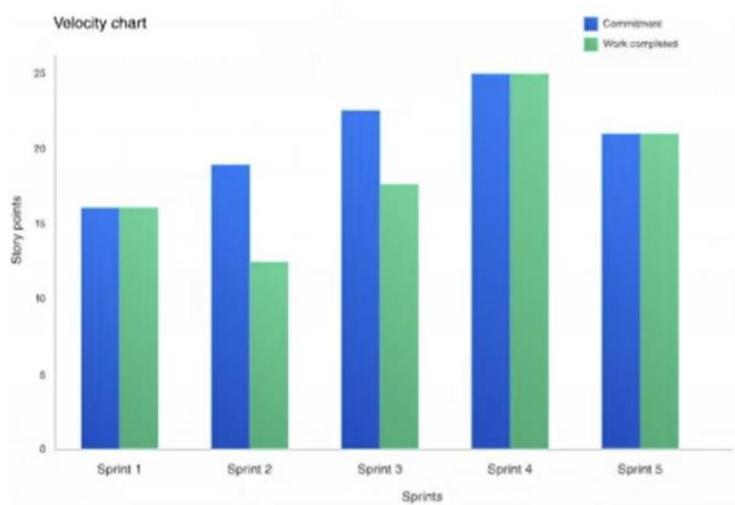
También se puede ver la estimación original, y cuando verdaderamente tomo. Acá por ejemplo la original era 15, y después tuve que agregar 4 en el sprint 8 y otras 4 en el sprint 9

Velocity Chart

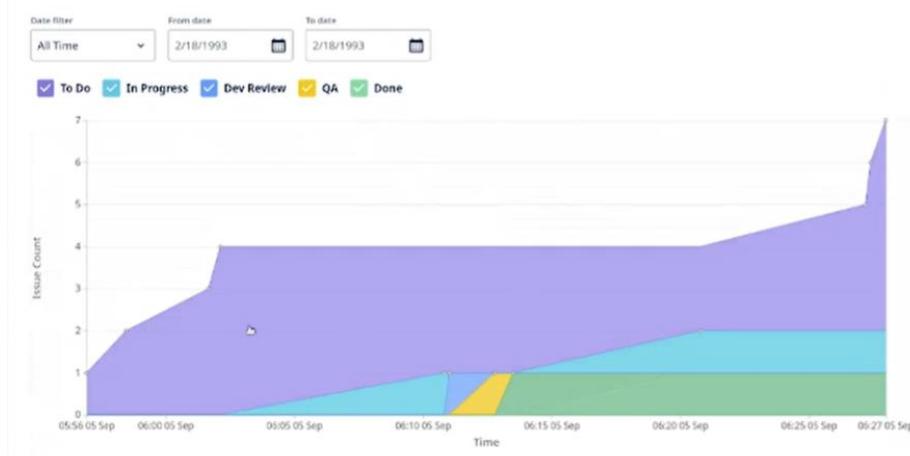
El Velocity Chart muestra la cantidad de trabajo completado (en story points) por el equipo durante cada sprint. Ayuda a medir el rendimiento del equipo y a predecir la capacidad futura.

Estructura del gráfico

- Eje X: Sprints (por ejemplo, Sprint 1, Sprint 2, Sprint 3, etc.).
- Eje Y: Story points completados.
- Barras: Cada barra representa los story points terminados en un sprint en comparación con los planificados.



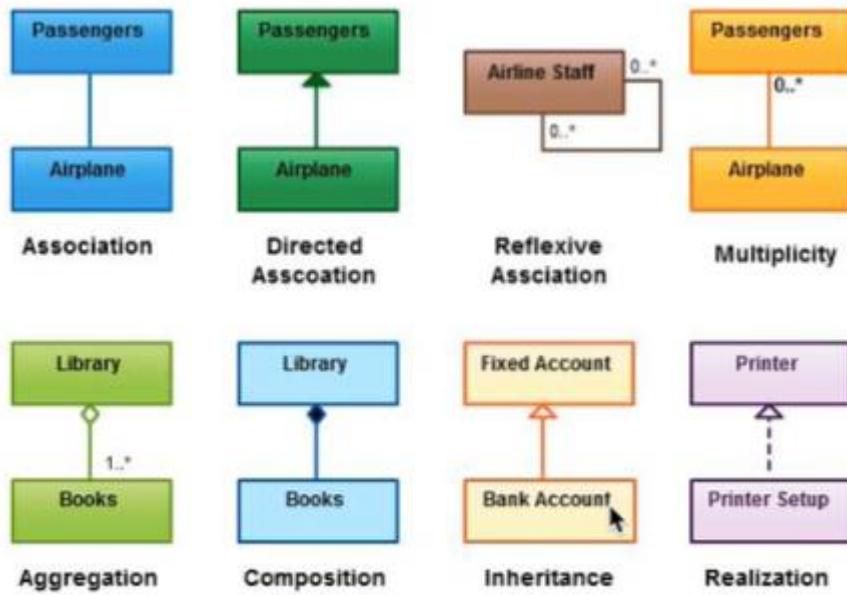
Cumulative Flow Diagram



Diagramas UML

Herramienta para representar visualmente diseños, arquitecturas y conceptos. Se usa para especificar y visualizar, facilitando la comprensión de ideas y conceptos.

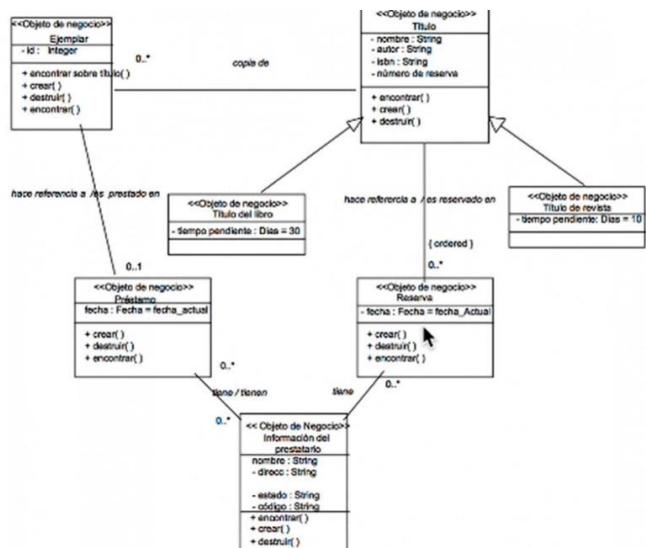
Tipos de relaciones



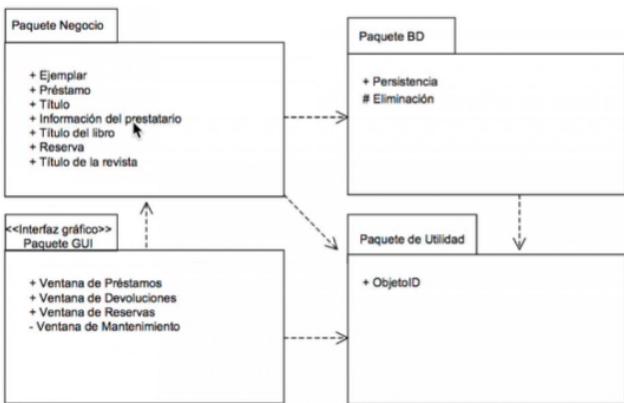
Relaciones en los diagramas de clase UML

Diagramas estructurales

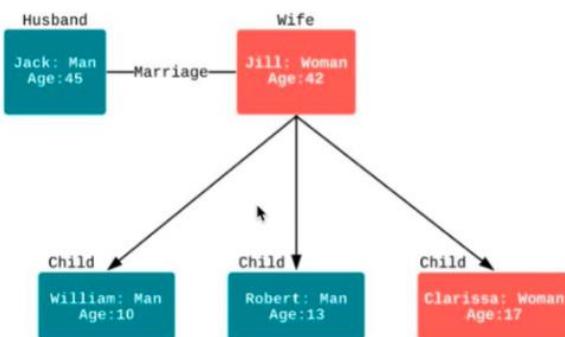
- Diagramas de clases



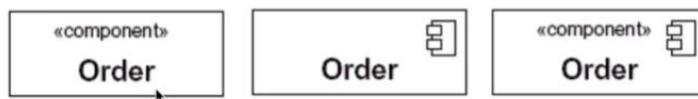
- Diagrama de paquetes – le da un orden a las clases/conceptos lógicos que armamos modelándolo según nuestro negocio.



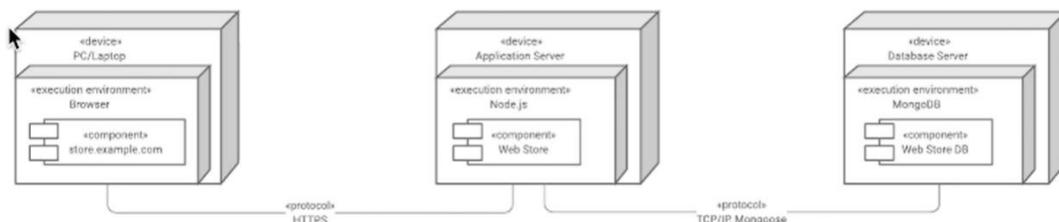
- Diagrama de objetos – mostrar objetos instanciados con ciertos valores y configuraciones especiales de instancias de estas clases



- Diagrama de componentes – como empaquetar los criterios lógicos; puede ser a nivel de archivo por ejemplo. Es algo físico que después puedo desplegar en una maquina.

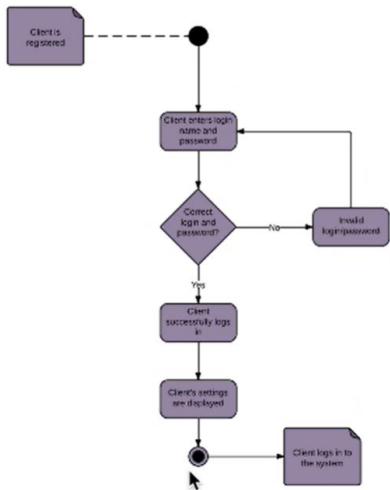


- Diagrama de despliegue – pensado para mostras modelos físicos (computadoras, teléfonos o algún hardware) para ver como se relaciona con otros nodos por ejemplo, a través de un protocolo http

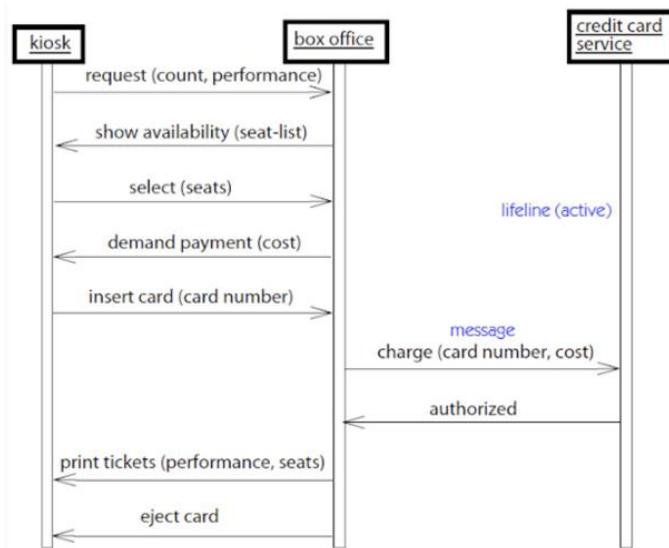


Diagramas de comportamiento

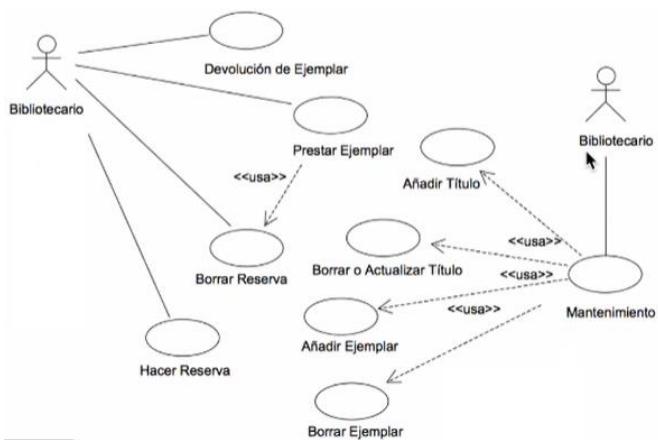
- Diagrama de actividades – diagrama de flujo desde un principio hasta un fin, por ejemplo de un procedimiento.



- Diagrama de secuencia – muestra la interacción entre objetos para resolver una funcionalidad



- Diagrama de caso de uso – nos da un panorama de los casos de uso



- Diagrama de máquina de estados – como cambia el estado a partir de una acción, hasta que vuelve a un estado fin o un estado inicial, etc.



ARQUITECTURA DE SOFTWARE

Conceptos o propiedades de un sistema en su entorno encarnado en sus elementos, relaciones y en los principios de su diseño y evolución.

La arquitectura de software representa la estructura/estructuras del sistema, que consta...

Son las decisiones que son importantes y difíciles de cambiar; parte del core que tengo que respetar.

¿Qué debemos tener en cuenta cuando escribimos un documento?

Lo principal es **quien** lo va a leer. Dependiendo a quien le voy a transmitir la idea es cómo me comunico y a qué nivel; si con más profundidad, si con mas o menos especificaciones técnicas, etc. El informe se debe apuntar a lo que espera el lector.

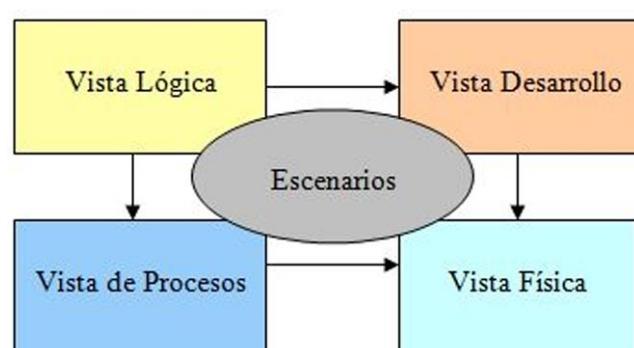
Cuando hablamos de arquitectura en un equipo de desarrollo de software, esa arquitectura le puede interesar a

- Otros desarrolladores
- Encargados del área de despliegue
- Clientes
- Analistas
- Diseñadores de productos
- Testers
- Líderes de proyectos
- Usuarios

Bajar la información en un único diagrama es difícil, porque dependiendo con quien me comunico que quiero mostrar y la profundidad/enfoque con el que voy a comunicarme.

Modelo de Vistas 4+1

Propone separar en vistas la distinta información de la arquitectura de nuestro sistema. en cada vista hondo con distinto detalle



1. Vista lógica

Apoya principalmente los **requisitos funcionales** – lo que el sistema debe brindar en términos de servicios a sus usuarios.

El sistema se descompone en una serie de abstracciones clave, tomadas (principalmente) del dominio del problema en la forma de objetos o clases de objetos. Aquí, se aplican los principios de abstracción, encapsulamiento y herencia. Esta descomposición no solo se hace para potenciar el análisis funcional, sino también sirve para identificar mecanismos y elementos de diseño comunes a diversas partes del sistema.

Acá usamos diagramas de clases, de secuencia, de componentes, por ejemplo.

Ejemplo 1:

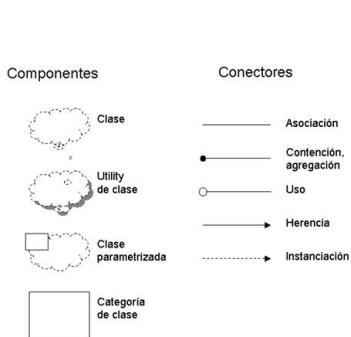


Figure 2: Notación para la vista lógica

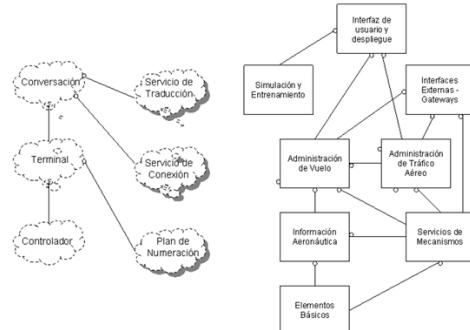
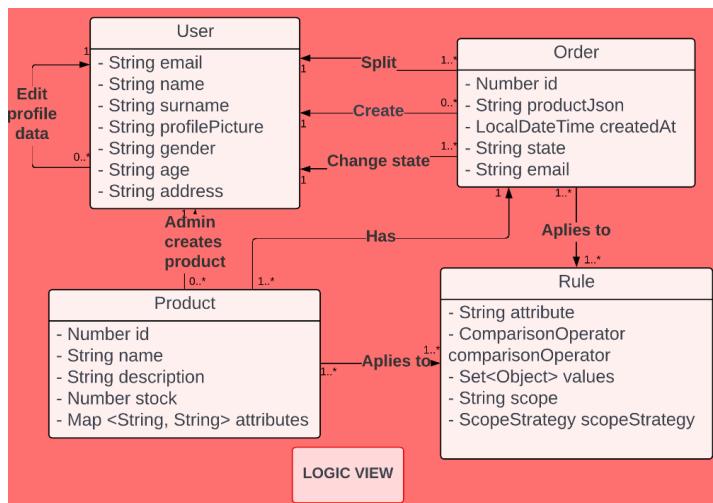


Figure 3: (a) Diagrama lógico del Télic PBX; (b) Diagrama de un sistema de control de tráfico aéreo

Ejemplo 2



2. *Vista de procesos*

Toma en cuenta algunos requisitos no funcionales tales como el rendimiento (performance) y la disponibilidad.

Se enfoca en asuntos de concurrencia y distribución, integridad del sistema, de tolerancia a fallas, etc. La vista de procesos también especifica en cuál hilo de control se ejecuta efectivamente una operación de una clase identificada en la vista lógica.

La arquitectura de procesos se describe en varios niveles de abstracción, donde cada nivel se refiere a distintos intereses. El nivel más alto la arquitectura de procesos puede verse como un conjunto de redes lógicas de programas comunicantes (llamados “procesos”) ejecutándose en forma independiente, y distribuidos a lo largo de un conjunto de recursos de hardware conectados mediante un bus, una LAN o WAN. Múltiples redes lógicas pueden usarse para apoyar la separación de la operación del sistema en línea del sistema fuera de línea, así como también para apoyar la coexistencia de versiones de software de simulación o de prueba.

Un proceso es una agrupación de tareas que forman una unidad ejecutable. Los procesos representan el nivel al que la arquitectura de procesos puede ser controlada tácticamente (como comenzar, recuperar, reconfigurar y detener). Además, los procesos pueden replicarse para aumentar la distribución de la carga de procesamiento, o para mejorar la disponibilidad.

Trata más sobre procesos que ocurren en memoria y con los recursos que tenemos. Cuenta la problemática a nivel de procesos del sistema. Indica cuántos procesos componen mi solución.

Ejemplo

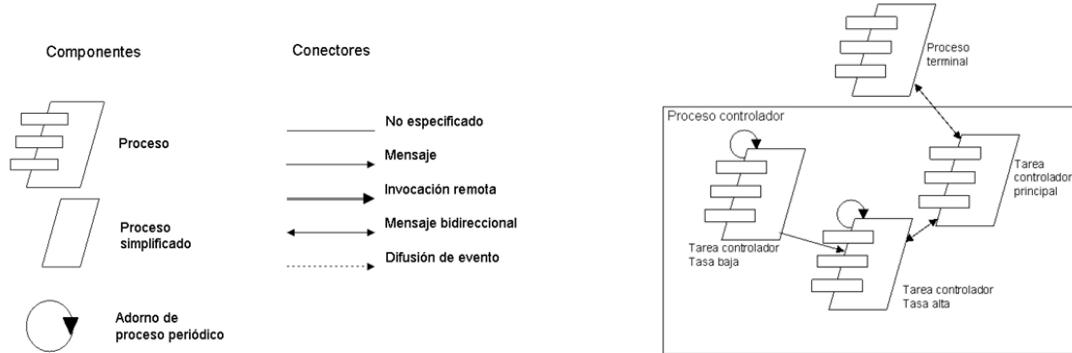
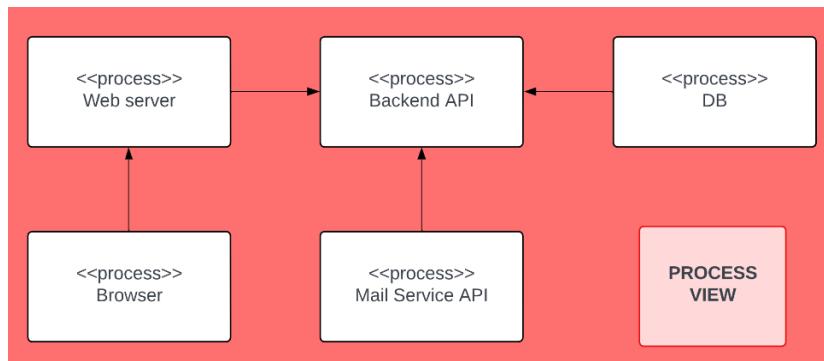


Figure 4: Notación para el diagrama de procesos

Figure 5: Diagrama (parcial) de procesos para Télic PBX

Ejemplo 2



3. Vista de desarrollo (o componentes)

Se centra en la organización real de los módulos de software en el ambiente de desarrollo del software. Esto es a nivel **físico**. **Hablo de qué componente voy a desplegar en cada nodo físico.**

El software se empaqueta en partes pequeñas; bibliotecas de programas o subsistemas, que pueden ser desarrollados por uno o un grupo pequeño de desarrolladores. Los subsistemas se organizan en una jerarquía de capas, cada una de las cuales brinda una interfaz estrecha y bien definida hacia las capas superiores.

La vista de drollo tiene en cuenta los requisitos internos relativos a la facilidad de desarrollo, administración de software, reutilización y elementos comunes, y restricciones impuestas por las herramientas o el lenguaje de programación que se use.

Depende de cómo empaquete eso me va a dar más o menos flexibilidad a la hora de desplegar, y cuantos más componentes tengo más tengo que gestionar la comunicación entre ellos.

Acá uno empieza a pensar acerca de todas las cosas lógicas que se ven en la vista lógica, si lo metemos en un único hard, en un byte code, en un Excel, o lo empaquetamos de diferentes maneras.

Ejemplo 1

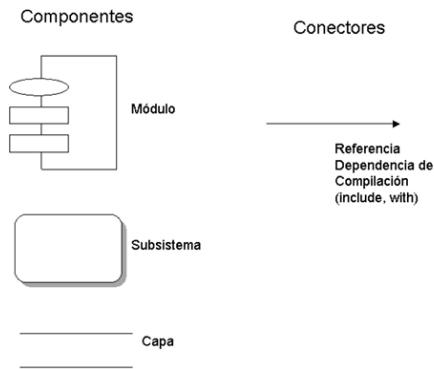


Figure 6: Notación para el diagrama de desarrollo

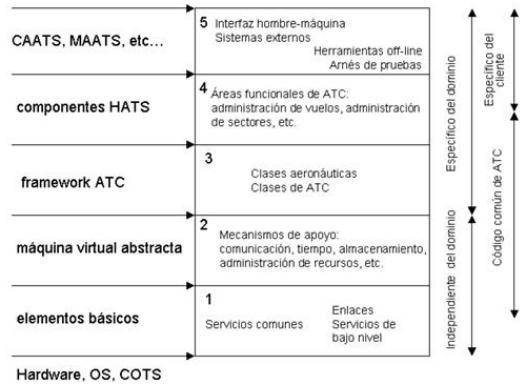
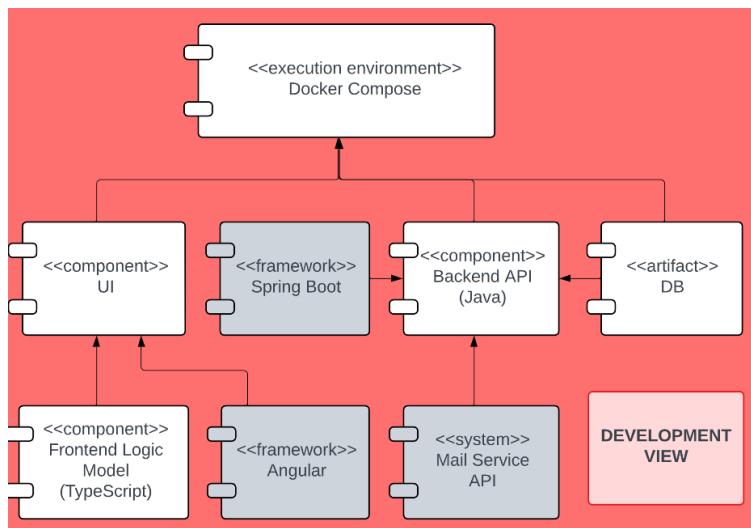


Figure 7: Las 5 capas del Sistema de Tráfico Aéreo de Hughes (HATS)

Ejemplo 2



4. Vista física (o de despliegue)

Toma en cuenta primeramente los requisitos no funcionales del sistema, tales como la disponibilidad, confiabilidad (tolerancia a fallas), rendimiento (throughput), y escalabilidad.

El software se ejecuta sobre una red de computadoras o nodos de procesamiento. Los variados elementos identificados – redes, procesos, tareas y objetos – requieren ser mapeados sobre los nodos.

Esperamos que diferentes configuraciones puedan usarse: algunas para desarrollo y pruebas, otras para mostrar el sistema en varios sitios para distintos usuarios. Por lo tanto, la relación del software en los nodos debe ser altamente flexible y tener un impacto mínimo sobre el código fuente.

Se trata de Nodos físicos; cuántas máquinas necesito para darle respuesta a la solución que estoy planteando. Cómo se conocen entre ellas, si hay un firewall en el medio, etc.

Ejemplo 1

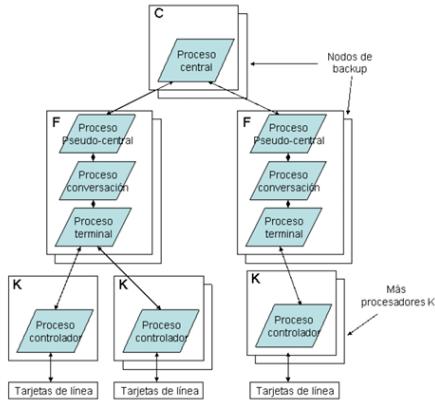
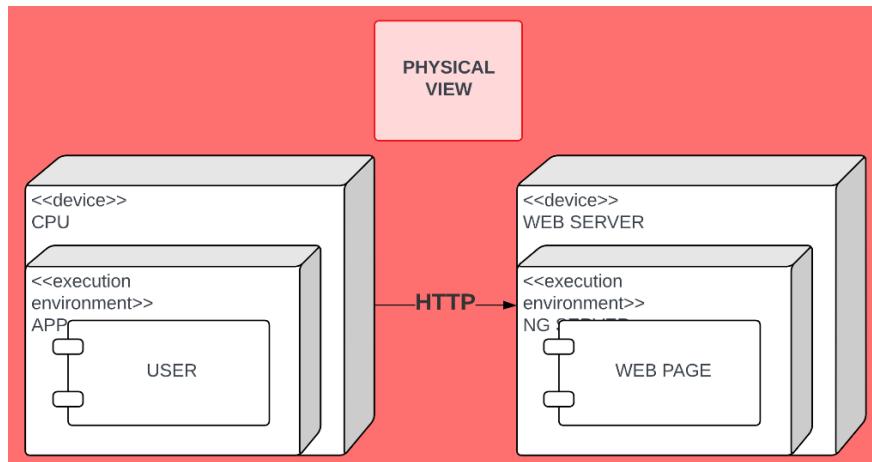


Figure 11: Diagrama físico para un PABX más grande incluyendo emplazamiento de procesos

Ejemplo 2



5. +1: escenarios y casos de uso – El Negocio

Los elementos de las cuatro vistas trabajan conjuntamente en forma natural mediante el uso de un conjunto pequeño de escenarios relevantes. Los escenarios son de alguna manera una **abstraccion** de los requisitos más importantes.

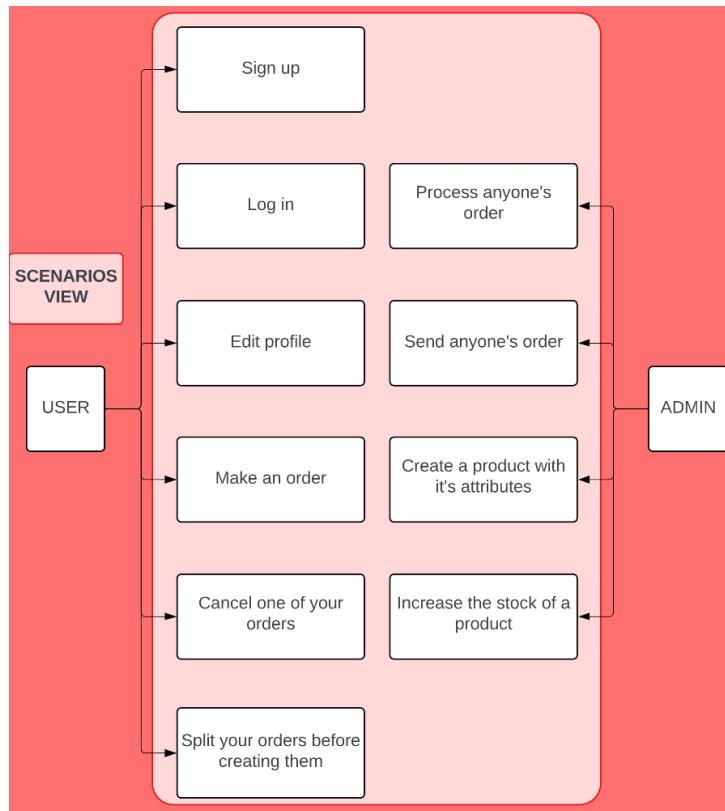
Sirve a dos propósitos principales:

1. Como guía para descubrir elementos arquitectónicos durante el diseño de arquitectura
2. Como un rol de validación e ilustración después de completar el diseño de arquitectura, en el papel y como punto de partida de las pruebas de un prototipo de la arquitectura.

Dadas ciertas problemáticas o escenarios, uno elige algunos para pensar la solución y arquitectura de un sistema. seleccionamos casos de uso que tengan cierta complejidad

En base a los escenarios, planteamos la arquitectura: ¿armamos una API Rest? ¿Que procesos voy a tener? ¿Cómo los voy a desplegar? Me baso en las vistas para diseñar eso.

Ejemplo



Conclusiones

Este modelo de vistas:

- Permite a través de diferentes vistas analizar distintas perspectivas del problema, focalizándose en el problema en cuestión.
- Concentra en un único documento las principales decisiones tomadas sobre el sistema
- Permite a nuevos integrantes del equipo entender la arquitectura del sistema y ubicarse dentro de la solución.
- Permite discutir con todos los stakeholders las distintas decisiones y validarlas en una etapa temprana.

Decisiones de arquitectura

A veces se había planteado una forma A pero al final se realizo de la forma B, pero no sabemos porque finalmente se tomo la forma B. a veces esas decisiones están bueno plasmarlas.

Las **decisiones de arquitectura** son decisiones de diseño que abordan requisitos significativos desde el punto de vista arquitectónico; se perciben como difíciles de hacer y/o costos de cambiar.

Estas decisiones son importantes de guardarlas y conocerlas; conocer el motivo por el cual se tomo dicha decisión en la arquitectura o diseño.

Por ejemplo, para el envío de emails se usa un SMTP y realiza el envío programáticamente, gestionado por nuestro backend, o se delega a un servicio de email provider de terceros, como Mailgun o Sendgrid.

Buscamos plasmar:

- Estado (propuesta: aceptada, deprecada, sustituida)
- Contexto
- Decisión
- Consecuencias

Requerimientos no funcionales

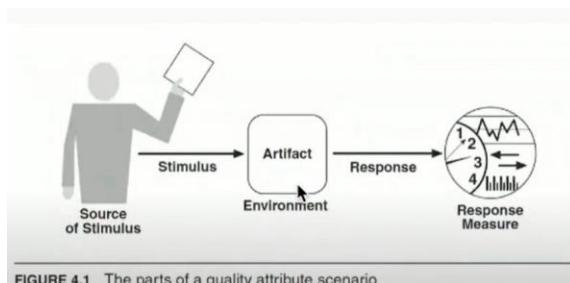
En general se trata de requerimientos que no tienen tanto que ver con la tarea que el cliente necesita, sino con, por ejemplo, requerimiento que lo necesito con tal calidad, o que funcione con X cantidad de requests por minuto, etc.

Atributos de Calidad

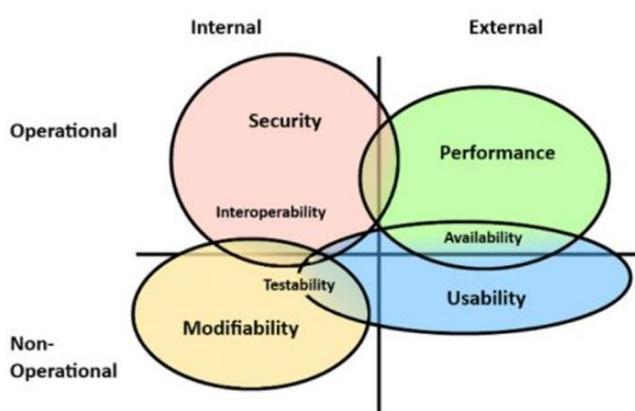
Es una medida o propiedad testeable de un sistema que es usada para indicar cómo el sistema satisface las necesidades de sus stakeholders.

Ejemplo: necesito que el tiempo-respuesta de este procesamiento sea menor a 20s.

Todos estos atributos de calidad tienen que poder medirse y testearse para ver si los estamos cumpliendo.



Se los puede clasificar como:



Dentro de los atributos de calidad, las normas ISO los clasifica como:



¿Por qué se rediseñan los sistemas?



Las consideraciones de calidad se desprenden de las necesidades del negocio y deben jugar un rol fundamental durante el ciclo de vida del desarrollo de software. No todo es funcionalidad en un sistema.

Los atributos de calidad deben ser considerados a lo largo de **todo el ciclo de vida del software** para tener éxito.

- No dependen solamente de la etapa de diseño (y/o arquitectura)
- No dependen slamente de la implantación a crear.

Ejemplos de atributos de calidad:

- Disponibilidad
- Reusabilidad
- Performance
- Robustez
- Flexibilidad
- Testeabilidad
- Interoperabilidad
- Usabilidad
- Mantenibilidad
- Integridad
- Portabilidad
- Confiabilidad

Los atributos de calidad pueden entrar en **conflicto unos con otros**:

- Seguridad vs confiabilidad
- Portabilidad vs performance
- Seguridad vs usabilidad
- Performance vs modificabilidad

El objetivo es evaluar cuantitativa y/o cualitativamente múltiples atributos de calidad:

- Consensuar una priorización de los atributos de calidad con los interesados involucrados
- Diseñar un sistema lo suficientemente bueno para todos los interesados

Disponibilidad (Availability)

¿Qué haría fallar al sistema? ¿Qué tan probable es que ocurra? ¿Cuánto tiempo lleva repararlo?

Se refiere a la **capacidad del sistema para estar en funcionamiento y accesible cuando se le necesita**: Qué capacidad tiene un sistema o aplicación de estar disponible y operativo para los usuarios durante un período de tiempo especificado.

En general se mide en horas, días o años. Algunos aspectos claves son:

Medición de la Disponibilidad Se mide generalmente en términos de un porcentaje, que representa la proporción de tiempo en el que el sistema está operativo en relación con el tiempo total.

Ejemplo: si un sistema está en funcionamiento durante 364 días al año, su disponibilidad sería del 99.73% (esto se calcula como $(364 / 365) * 100\%$).

Factores Clave: La disponibilidad del software se ve influenciada por varios factores, que incluyen:

- Diseño Robusto
- Gestión de Recursos
- Mantenimiento y Actualizaciones
- Supervisión y Detección de Fallos
- Recuperación ante Desastres

Importancia de la Disponibilidad: Negocios, Usuarios Finales , Seguridad

Ejemplos de Estrategias para Mejorar la Disponibilidad: Balanceo de carga, Replicación de Datos, Respaldo y Recuperación, Escalabilidad.

La disponibilidad es esencial para garantizar que los sistemas de software cumplan con las expectativas de los usuarios y las necesidades del negocio. Por lo tanto, los ingenieros de software deben diseñar y desarrollar aplicaciones teniendo en cuenta la disponibilidad desde el principio, implementando estrategias para mitigar fallos y garantizar la continuidad del servicio.

Performance

Corresponde a los **tiempos de respuesta de la aplicación en relación a las funcionalidades** o actividades soportadas por la misma. Se consideran dos formas principales para medir el rendimiento de una aplicación:

- **Latencia:** Tiempo dedicado a responder a un evento.
- **Capacidad:** El número de eventos que pueden ocurrir en un tiempo determinado.

Interoperabilidad

Mide la **capacidad de intercambio de información de la aplicación con otros sistemas** o con el entorno donde opera. Una aplicación bien diseñada facilita la integración con otros sistemas.

Para mejorar la interoperabilidad, es conveniente utilizar interfaces externas bien diseñadas, normas de intercambio y estándares, entre otras.

Usabilidad

La usabilidad se puede ver a través de:

- **Comprendibilidad:** este atributo refleja que tan fácil es para el usuario comprender el sistema, que conocimientos previos requiere el usuario para poder trabajar con el software
- **Fácil uso / Eficiente:** Que permita realizar las operaciones de manera rápida y efectiva
- **Fácil de Recordar / Intuitiva / Estándar**
- **Atractividad / Agradable / Cómoda**

Seguridad

Permite medir la **vulnerabilidad de las aplicaciones a ataques accidentales o maliciosos**, versus la posibilidad de defensa del sistema ante pérdidas o robo de información estratégica y valiosa para la organización. Estos ataques pueden ser tanto externos como internos

Algunos métodos de seguridad son Autenticación, autorización, encriptación de datos, auditoría, entre otros.

- Capacidad de detección de ataques de denegación de servicio (DDoS), y respuesta ante éstos.
- Restricciones de acceso de usuarios de acuerdo a las políticas de autenticación y autorización.
- Prevención de la inyección de consultas SQL
- Encriptación de claves, contenidos y datos empresariales.
- Conexión segura

Escalabilidad

Es la **capacidad de manejar la carga de trabajo de la aplicación sin afectar el rendimiento de la misma**, es la posibilidad de crecimiento sin perjudicar su funcionamiento operativo.

Cómo mejorar la escalabilidad:

- **Vertical:** Para crecer, se agregan más recursos físicos a la infraestructura que soporta al aplicativo, tales como memoria, almacenamiento en disco, procesador o capacidad de cómputo, ancho de banda, entre otras, para un aplicativo.
- **Horizontal:** Se incrementa el número de computadores para dividir la carga de trabajo de la aplicación.

Entre los indicadores claves para medir la escalabilidad se encuentran:

- Si el sistema permite el escalamiento vertical o distribución de la carga de trabajo en distintas computadoras.
- El tiempo necesario para aumentar el escalamiento.
- Las limitaciones de escalamiento en la infraestructura operativa: número de servidores máximo, memoria, discos, o capacidades de la red.
- Posibilidades de escalamiento: incremento en el número de transacciones o carga de trabajo.

Patrones de Arquitectura

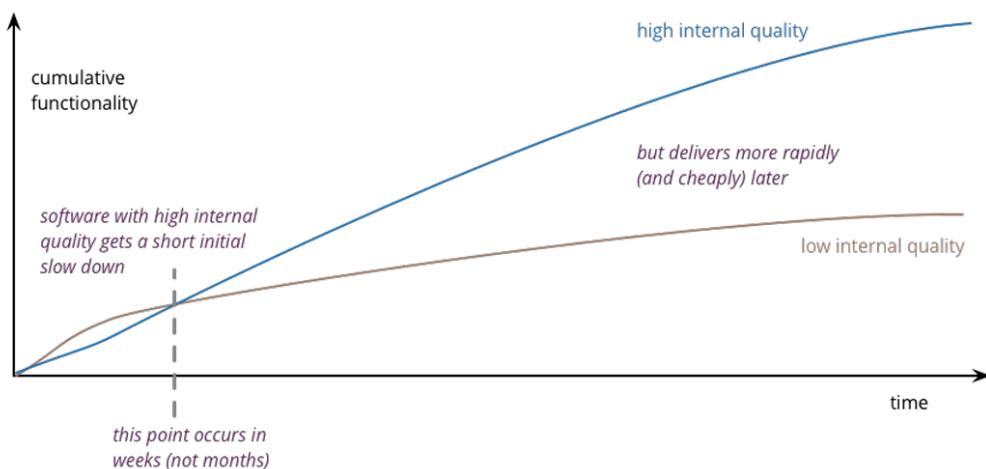
Cuando hablamos de arquitectura nos referimos a los bloques fundamentales de nuestro proyecto. Se trata de cómo vamos a estructurar la solución

Dentro de cada uno de los bloques vamos a diseñar código. Para el código cumpla ciertos principios, como los SOLID, es donde aplican los principios y patrones de diseño.

La **arquitectura de software** representa el conjunto de decisiones de diseño significativas que dan forma a la forma y función de un sistema, donde lo significativo se mide por el costo del cambio.

- Lo que le da forma al sistema
- Decisiones significativas del diseño
- Se mide en costo de cambio
- Representa la división del sistema en componentes
- Relacionada a la comunicación entre componentes

Why does architecture matter?



Un sistema con una arquitectura, prolja, ordenada, con las partes bien definidas tiene una calidad interna alta. Para las low internal quality, inicialmente puede entregar mayor funcionalidad, pero existe un punto de inflexión en el tiempo donde decae fuertemente. Esto lo podemos ver en que, en un sistema bien estructurado, es fácil ir implementando nuevos features.

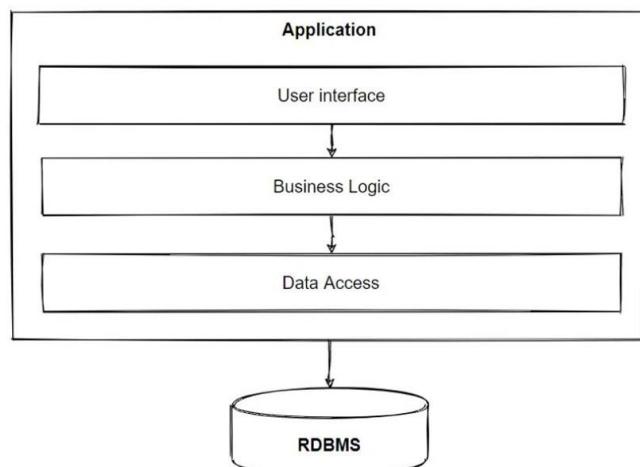
PATRONES DE ARQUITECTURA

1. Layers

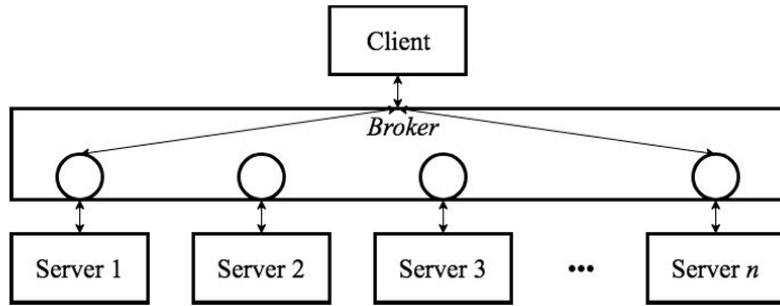
Propone estructurar el sistema en un conjunto de capas que se comuniquen entre sí, donde una capa se comunica únicamente con la siguiente, y solamente con la interfaz de la siguiente capa, no se mete en la implementación.

Dado que yo acepto que una forma posible de dividir mi sistema es en capas, puedo definir, como arquitecto, cuantas capas voy a tener y como se relacionan entre si. Acá hay un ejemplo

1. Interfaz con el usuario: capa con la que interactúan los usuarios (usando un frontend o llamando a una API por ejemplo), osea como le presento los datos al usuario
2. Capa lógica: donde esta toda la lógica del sistema, casos de uso de mi sistema
3. Capa de acceso a los datos (capa que se encarga de acceder a la BDD, a conectarse): controla como se persiste la data en mi aplicación.



2. Broker



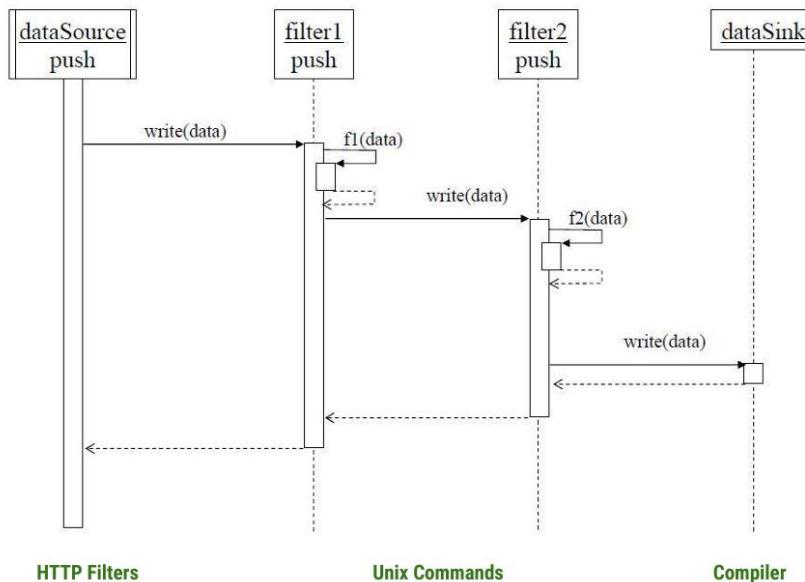
Es un intermediario que separa los costos y desacopla el código del cliente con el del servidor.

El cliente le hace la petición al bróker y ese mismo decide qué bróker resolverá esa request. Hay distintas estrategias para decidir qué servidor lo resolverá; una es round robin (la 1ra consulta el 1er servidor, la 2da el 2do... etc.), otra es aleatoria, etc.

El bróker no necesariamente tiene la misma interfaz que el server.

Una idea análoga es el trabajo entre interfaz de usuario – cluster – nodos (spark por ej)

3. Pipe & Filters

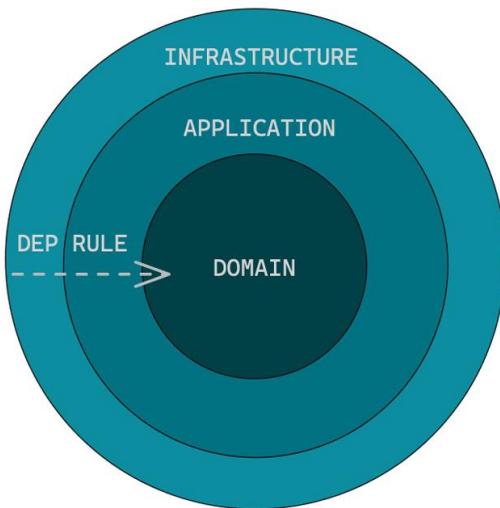


La idea es armar una cadena de eslabones para resolver una consulta determinada (ejemplo: ETL – Extract, Transform and Load: it is a three-phase process where data is extracted from an input source, transformed (including cleaning), and loaded into an output data container.)

Tengo una fuente de datos y un conjunto de filtros que procesan esa data. Luego, tengo un dataSink que termina de juntar esa data.

4. Arquitectura Hexagonal

Hexagonal Architecture (Ports & Adapters)



Patrón de puertos y adaptadores: los usamos para todos los inputs y outputs de la app

Suponiendo que el core de la app está en el centro, el objetivo es que para todo input/output usemos un puerto (que no es más que una abstracción; es una manera que la app interactúe con el exterior/otros sistemas, sin tener que saber con quién está interactuando).

Ejemplo: tengo un puerto de input y output. No importa a donde estos escribiendo, si sea una bdd o un filesystem; siempre y cuando tenga una manera de escribir/leer, eso es lo único que le importa a la aplicación.

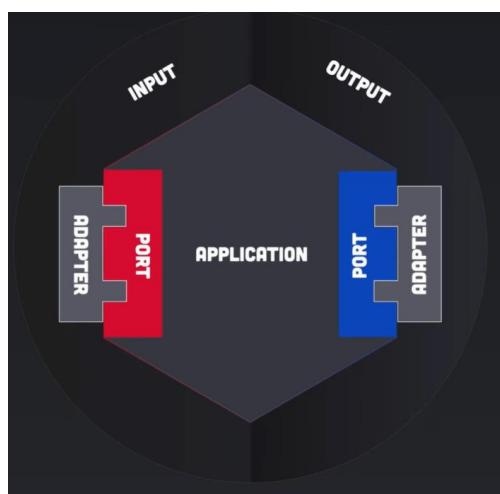
Puerto: custom interface a la aplicación.

Lleva al decoplamiento a otro nivel.

Adaptadores: donde la lógica core sucede, que toma el output de mi aplicación y lo convierte en algo que pueda ser usado por una fuente externa.

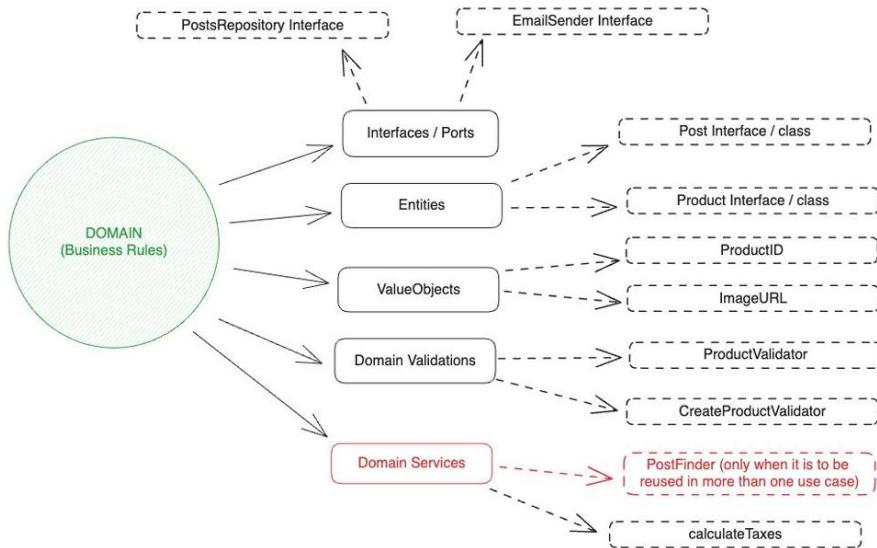
Lo que importa de una aplicación son los casos de uso, que encapsulan la lógica del sistema. el resto: Como me comunico con el usuario, como lo comunico; el lo denomina **adaptadores**, meto algo que adapte a lo que yo tengo.

Lo importante es el dominio. Alrededor de eso están los casos de uso (Application) y todo el resto es infraestructura. La regla es que la flecha siempre va para adentro.



No importa que cambio del adaptador, la app con sus puertos nunca cambia.

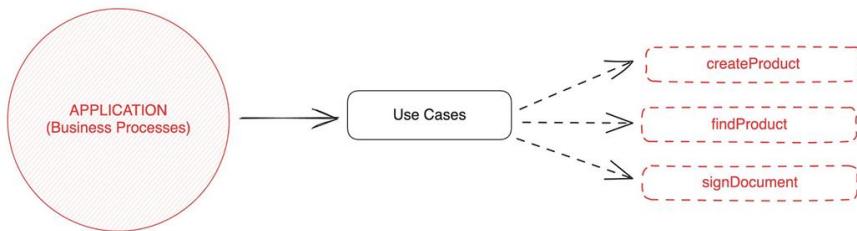
Capa de dominio



El dominio son las entidades + reglas de negocio.

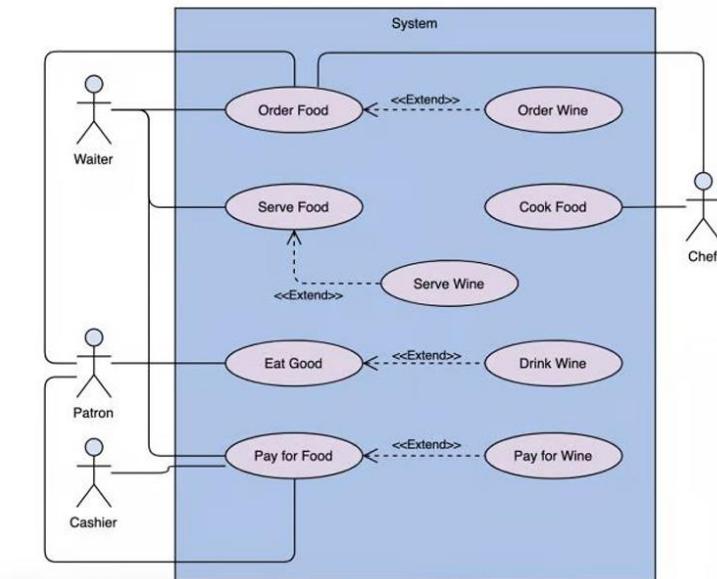
Las interfaces, viven dentro del dominio y son los puertos (port). La implementación de esa interfaz vive en la infraestructura, y son los adapters.

Capa de aplicación

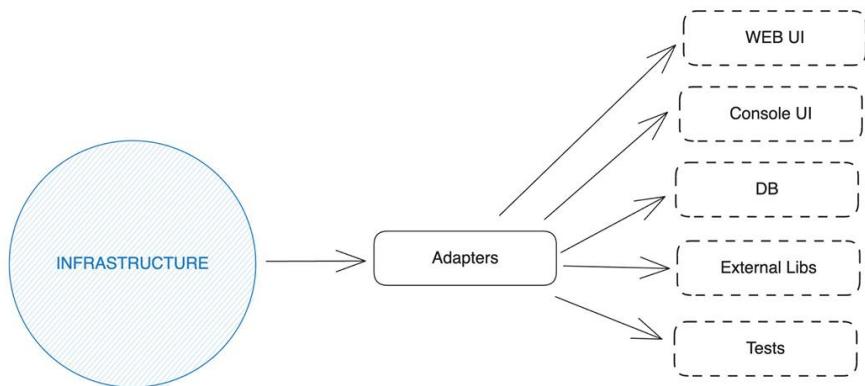


Todas las interacciones representan una operación que resuelve mi sistema (ese firmarDocumento, por ejemplo es lo que denominamos caso de uso)

Ejemplo de diagrama de Caso de uso:



Capa de infraestructura



Acá están los adaptadores de los ports; cómo lo implemento

Ejemplo de caso de uso:

Cada función representa un caso de uso, y cada uno es una clase

```
export function createUser(user: User) {
  return db.collection('users').add(user);
}

export function findUser(userId: string) {
  return db.collection('users').doc(userId).get();
}

export function updateUser(userId: string, user: User) {
  return db.collection('users').doc(userId).update(user);
}

export function deleteUser(userId: string) {
  return db.collection('users').doc(userId).delete();
}
```

Yo podría tener los mismos casos de uso, pero en vez de conectarme con la base de datos, podría conectarme con un endpoint que lo resuelva:

```

export async function createUser(user: User) {
  const response = await fetch('https://myapp.com/users', {
    method: 'POST',
    body: JSON.stringify(user),
  });
  return response.json();
}

export async function findUser(userId: string) {
  const response = await fetch(`https://myapp.com/users/${userId}`);
  return response.json();
}

export async function updateUser(userId: string, user: User) {
  const response = await fetch(`https://myapp.com/users/${userId}`, {
    method: 'PUT',
    body: JSON.stringify(user),
  });
  return response.json();
}

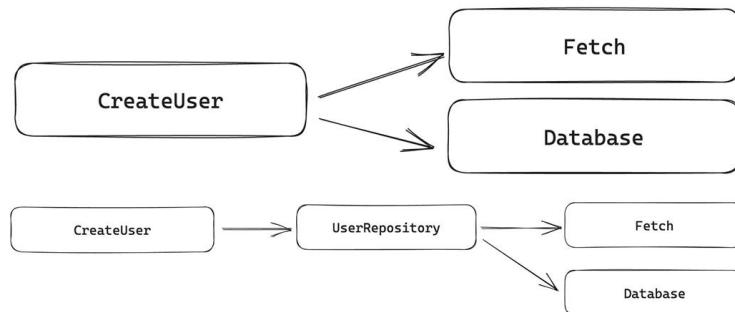
export async function deleteUser(userId: string) {
  const response = await fetch(`https://myapp.com/users/${userId}`, {
    method: 'DELETE',
  });
  return response.json();
}

```

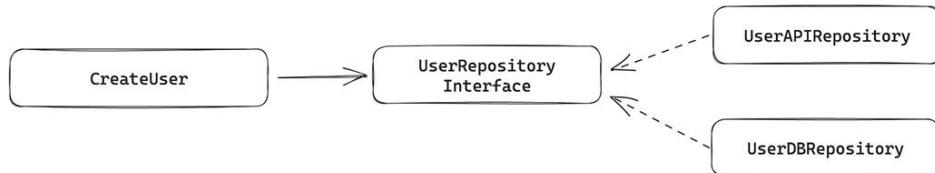
Testing?

5. soliD (Dependency Inversión Principle)

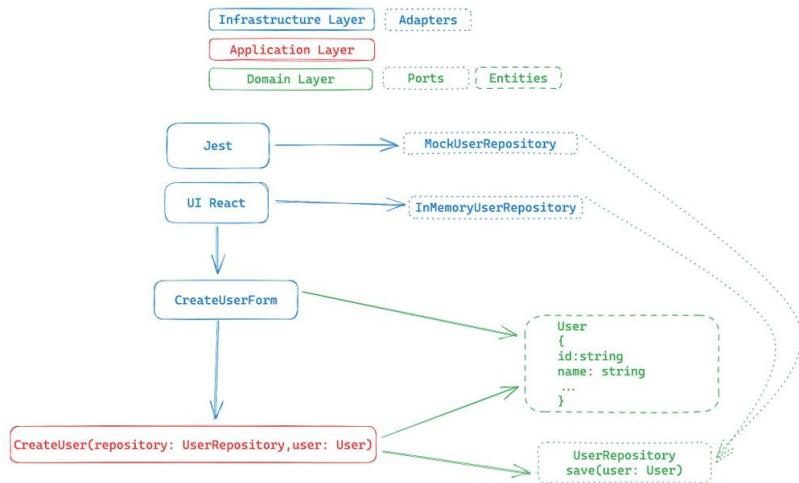
DIP: Los módulos de alto nivel no deberían depender de los de bajo nivel. Ambos deberían depender de abstracciones.



Además, las abstracciones no deberían depender de los detalles; los detalles deberían depender de las abstracciones



Workflow



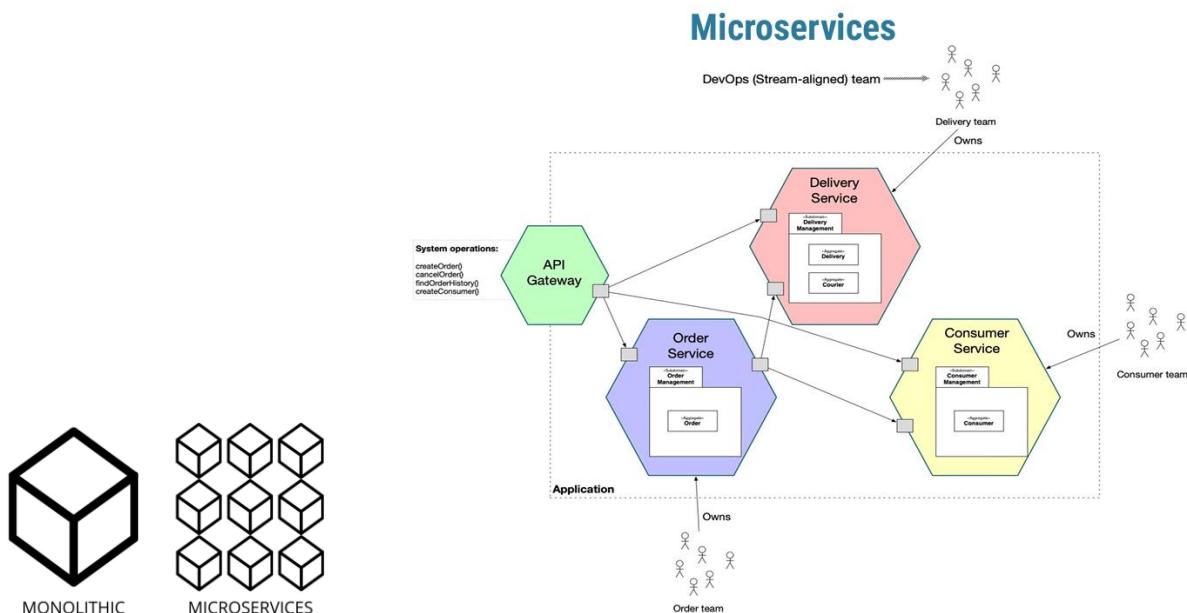
Estructura del proyecto:

```

modules
  users
    application
      createUser.ts
      listUsers.ts
      removeUser.ts
    domain
      User.ts
      UserId.ts
      UserRepository.ts
  infrastructure
    InMemoryUserRepository.ts

```

MICROSERVICIOS



Lo partimos en servicios mas pequeños para separar las funcionalidades de mi sistema, donde cada servicio responde a una problemática distinta por ejemplo. Eso me permite desacoplar el diseño

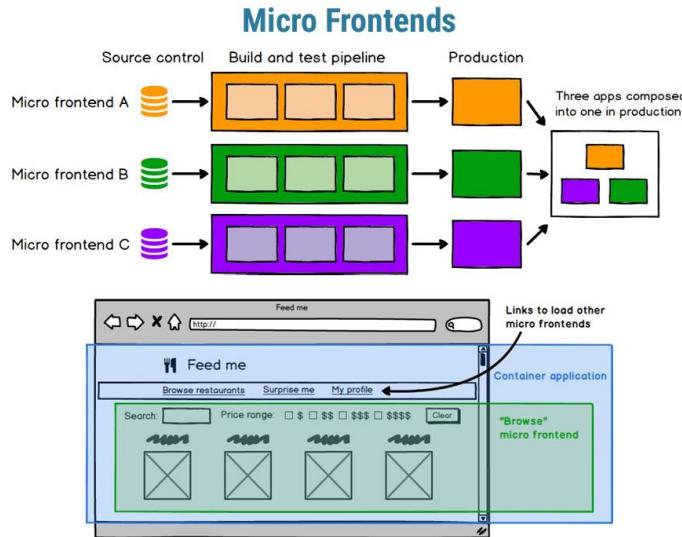
para cada microservicio. Por ejemplo puede ser que para un servicio sea mas eficiente implementarlo con un lenguaje orientado a objetos, y otro con paradigmas funcionales por ejemplo. Por otro lado, esto me permite escalar mi sistema fácilmente y agregar nuevos servicios

Presenta ciertas desventajas, como dificultades para detectar errores al tener muchos servicios. Por otro lado, tenemos que implementar la comunicación entre los servicios (una red, un sistema de mensajería, etc.). me tengo que hacer cargo de la implementación de la comunicación entre los servicios, manejar errores de comunicación, manejar la concurrencia, problemas de visibilidad y permisos, etc.

En general cada microservicio tiene internamente su arquitectura, diseño, etc.

El API Gateway es el que se encarga de delegar la consulta al microservicio encargarlo de resolverlo. No hay comunicación directa con los servicios sino que esta el Gateway de por medio.

MICRO FRONTENDS



La misma idea de microservices pero aplicado al front end. Divido las responsabilidades del front end en pequeñas partes.

Modelos de Dominio

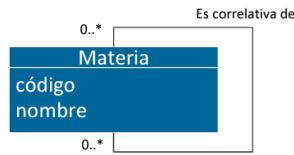
Es una representación visual de los tipos de entidades del dominio del problema y sus relaciones, que el sistema a construir necesita conocer.

Es un modelo **conceptual, descripto con un diagrama de clases**. Como es conceptual, no tiene que llevar muchos detalles técnicos.

- Se marcan las relaciones entre clases con asociación. (puede haber mas de una asociación entre dos entidades):

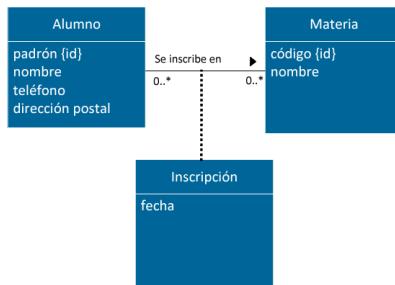


Tambien puede ser que la asociación sea unaria, pero igualmente hay que poner una multiplicidad para cada extremo de la unaria:



- Se suele acompañar la flecha con la descripción de la relación que las une
- Se ponen las multiplicidades
- Solo se ponen los atributos, no los métodos de las clases

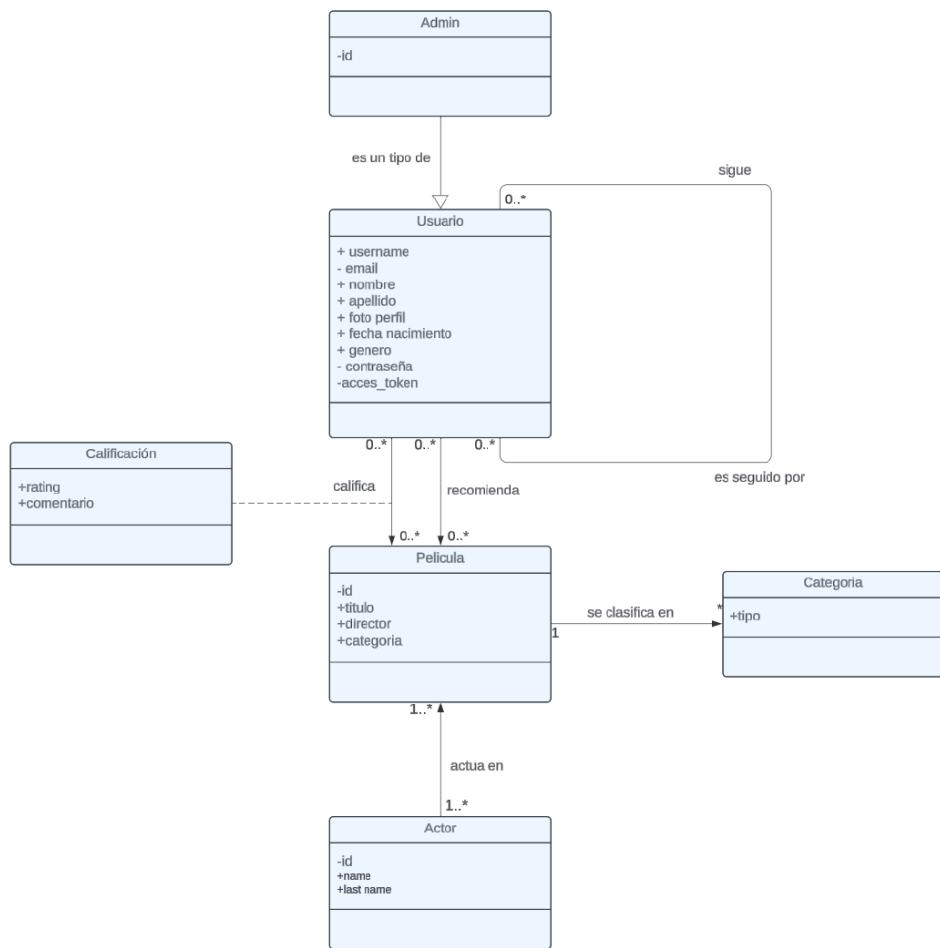
Existen también los **entidades/objetos asociativos**: objetos que se comportan simultáneamente como objetos y asociaciones. Estos dependen de una única relación. **No tienen identificador porque no son objetos fuertes**; su idd queda definida por su relación entre otros dos objetos.



Consejos

- ✓ Es un modelo conceptual, por lo que no se deben incluir aspectos vinculados a la implementación.
- ✓ Es preferible expluir las relaciones de composición y las ternarias, porque son mas complejas
- ✓ Hacer buen uso del lenguaje de modelado.
- ✓ No confundir atributos con relaciones: Si es un número, o un texto, probablemente no sea una entidad del dominio, simplemente un atributo más.
- ✓ Incluir notas aclaratorias.
- ✓ Complementar el modelo con un glosario o diccionario de datos.
- ✓ Si hay una única instancia de una entidad, probablemente no lo sea.
- ✓ No puede haber entidades sin atributos.

Ejemplo:



API REST

REST NO es un protocolo, NO es un estandar y NO es una biblioteca

Es una **interfaz de programación de aplicaciones que se ajusta a la arquitectura REST**. Significa Representational State Transfer. Utiliza estándares existentes como HTTP y maneja una comunicación cliente-servidor.

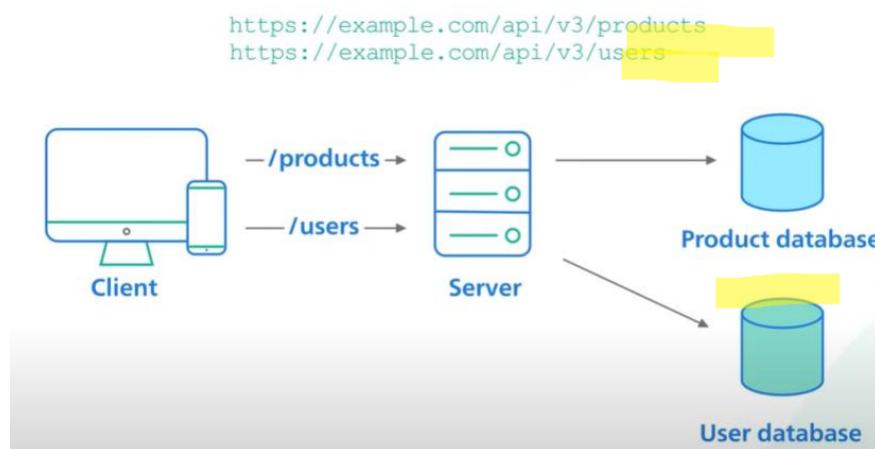
Características

- Usado en **arquitecturas cliente-servidor**
- **Stateless** (el servidor no guarda info de la sesión del cliente. Cada request y response es independiente de otras, por lo que las aplicaciones web se vuelven fáciles de escalar.). Se promueve el uso de tokens para manejo de seguridad
- **Cacheable** (capacidad etiquetar las rutas a los servicios pueden manejar un tipo de cacheo para minimizar interacciones con el server y así reducir el ancho de banda usado y la latencia)

Cache: quiero consultar X cosa. La 1ra vez llamo al servicio y obtengo los datos. Cuando cacheo almaceno la info (cuando es seguro que son datos fijos) que es cacheable. La próxima vez que llame la misma consulta no vuelvo a buscar el dato sino que ya lo tengo cargado.

Existen sistemas/mecanismos intermedios entre el cliente y el servidor, que no afectan como se consumen los servicios, pero cachan cierta información en esos puntos intermedios para reducir la cantidad de consultas directas al servidor.

- **Expone recursos a través de la URI**. Las URIs diferencian distintos tipos de recursos de un servidor. Por ejemplo:



Un cliente interactua con un recurso haciendo un request al endpoint del recurso a través de HTTP. Un request tiene el siguiente formato: **verb**: le dice al servidor qué queremos hacer con el **resource**



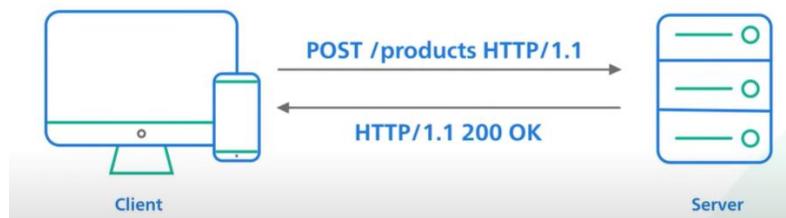
- Usa explícitamente los verbos HTTP

POST	→	CREATE
GET	→	READ
PUT	→	UPDATE
DELETE	→	DELETE

de acá sale CRUD

En el body de estos requests, puede haber un **HTTP request body** opcional que contiene un payload de datos, en general encodeado en JSON.

El servidor recibe el request, la procesa y le envía al cliente una respuesta:



Códigos de respuesta:

- ✓ 200 level: success
- ✓ 400 level: hay algún problema con la request (por ejemplo de sintaxis)
- ✓ 500 level: hay un problema a nivel servidor

El response body es **opcional**, y puede contener el data payload.

- Compresión

Las APIs suelen retornar representaciones en varios formatos, entre ellos formato plano, XML, HTML, JSON, los cuales pueden ser comprimidos para ahorrar ancho de banda sobre la red.

Accept-Encoding

Ejemplo de cómo el cliente informa que mecanismos soporta:

```
Accept-Encoding: gzip,compress
```

Content-Encoding

Ejemplo de cómo el server informa que mecanismo usó para encripción:

```
Content-Type: text/html
Content-Encoding: gzip
```

- Son navegables

URIs: Uniform Resource Identifier

Son identificaciones univocas de **recursos** con cadenas de caracteres. Permite identificar los recursos por clase o tipo. Además, permiten la distinción entre recursos principales y subordinados.

Por convención, se usan sustantivos en plural. **NO verbos**.

Ejemplos

Recurso: clientes

- **/clientes** representa todos los clientes
- **/clientes/1** representa al cliente con id 1
- **/clientes?nombre=juan** representa a los clientes con nombre juan
- **/clientes/1/compras** representa a las compras del cliente 1

Antes del 1er / suele ir la base del servidor

Y si las compras fueran recursos primarios:

Recurso: compras

- **/compras**
- **/compras?cliente=1**

HTTP: Verbos y Códigos

- **GET**: solicita una representación de un recurso específico
- **POST**: se utiliza para enviar una entidad a un recurso en específico
- **DELETE**: borra un recurso en específico
- **PUT**: reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición
- **PATCH**: aplica modificaciones parciales a un recurso (a diferencia de PUT)
- **OPTIONS**: es utilizado para describir las opciones de comunicación para el recurso de destino

Códigos de respuesta:

- 100 level: información (hold on)
- 200 level: éxito (hago lo que me pediste)
- 300 level: redirección (salí de acá)
- 400 level: Client error (te equivocaste vos)
- 500 level: Server error (me equivoque yo)

200 - OK	408 - Request Time-Out
201 - Created (con el location en el header)	409 - Conflict
400 - Bad Request	422 - Unprocessable Entity
401 - Authorization Required	500 - Internal Server Error
404 - Not Found	502 - Bad Gateway
405 - Method Not Allowed	504 - Gateway Time-Out

Principios de Diseño de las APIs REST

Least Privilege: Tener el menor privilegio requerido para hacer las acciones.

Fail-Safe Defaults: Por defecto no tener acceso a los recursos

Complete Mediation: El sistema debe validar los permisos de acceso a todos los recursos

Keep it Simple

Https

Password Hashes: (PBKDF2, bcrypt, y scrypt)

Never expose information on URLs: Usernames, passwords, session tokens, y API keys deberían no aparecer en la URL para evitar ser logueadas en los logs de web server logs

Considerar agregar Timestamp en los requests.

Validación de los parámetros de entrada

Monitoreo de transacciones sospechosas

Cantidad de requests por IP o por token/JWT/user para evitar problemas de denegación de servicio, o simplemente controlar o reducir el uso excesivo que puede bajar la performance de la API en general.

Limitación de velocidad, o tiempos de demora agregados entre request y request para ciertos casos, ayuda a reducir las solicitudes excesivas que ralentizarían la API, ayuda a lidiar con llamadas / ejecuciones accidentales y monitorea e identifica de manera proactiva una posible actividad maliciosa.

APIs pagas como las de google por ejemplo permiten configurar límites de uso, tarifa, para evitar sorpresas ante un mal uso o bug que genere por error multiples llamadas a la API.

Tipos de Autenticación y Autorización de las API REST

la autenticación verifica la identidad del usuario que quiere acceder a un recurso. Por otro lado, la autorización valida si efectivamente el usuario tiene el permiso para acceder al mismo o realizar alguna función.

La API Rest pone un punto de comunicación para 3ros con nuestra capa de servicios, así que si o si tiene que tener una autenticación.

- Basic auth

Encodea en Base64. Para un usuario y contraseña, arma un string usuario:contraseña lo cual se encodea en base64. Lo encodeado lo manda en un header que dice Authorization : lo encodeado.

- Base64 encoding
 - user: **fiuba**
 - pass: **k@X4R\$KFEbCn**
 - plain-auth: **fiuba:k@X4R\$KFEbCn**
 - Authorization: **Zml1YmE6a0BYNFIkS0ZFYkNu**

En general esto se usa en conjunto con otro mecanismo de seguridad, como HTTPS/SSL porque B64 es fácilmente decodificable.

- API Keys

Es un token que el cliente provee cuando hace la llamada. Hay varios métodos:

Via queryString:

1. **GET /something?api_key=abcdef12345**

Como header:

1. **GET /something HTTP/1.1**
2. **X-API-Key: abcdef12345**

Como cookie:

1. **GET /something HTTP/1.1**
2. **Cookie: X-API-KEY=abcdef12345**

API Keys se supone que es secreta y que solo el cliente y el servidor la conocen. Solo se debe usar en conjunto con otro mecanismo de seguridad como HTTPS/SSL

- Bearer Authentication

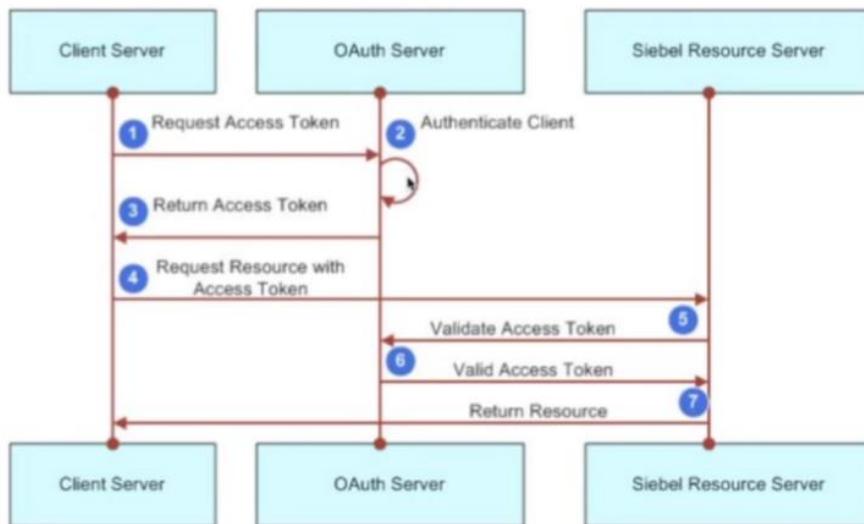
Utiliza tokens de seguridad llamados Bearer (da acceso al portador del token). Se envía en un header de Authorization

1. Authorization: Bearer <token>

Esto genera un servicio que consulta 1 vez si el usuario esta autorizado, en caso de que si, le envia un token con vencimiento y eso autoriza al usuario a tener acceso a ciertos servicios mientras que el token no este vencido.

- OAuth

Es un protocolo de autenticación. Consiste en delegar la autenticación de usuario al servicio que gestiona las cuentas (osea a un tercero; por ejemplo Google), de modo que sea éste quien otorgue el acceso para las aplicaciones de terceros.



Yo me autentico en por ejemplo Google con usuario y contraseña y me genera un token. Luego, cuando yo quiera hacer algo, el servicio que yo quiero le consulta al OAuth a ver si el token que generó es valido o no.

OAuth2 provee un flujo de autorización para aplicaciones web, móviles y también a programas de escritorio

- JWT

En las otros opciones, el server siempre tenía una llamada a la base x cada vez que se usaba la API Rest, porque tiene que consultar si el usuario esta autenticado/el token es valido.

El token se genera en un primero paso:

- user: **fiuba** / pass: **k@X4R\$KFEbCn**
- POST -> /token { "user": "fiuba", "pass": "k@X4R\$KFEbCn" }
- Response:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1aWQiOjEyMzQ1Njc4OTAsIm5hbWUiOiJKb2hulE
RvZSlslnByb2YiOiJPV05FUislm9yZyl6ODM0NzUyM30.2PWhCiy6sgDeBhGFbC1Ws1wl0Ggy7e
Y-44uey_aR0eo

Este token lo que tiene es un JSON con una estructura adentro, firmado con una clave privada que solo el servidor conoce y el manda.

El servidor chequea si ese token es el que el mandó (no se debe haber cambiado la firma) y si lo es, puede chequear en JSON dentro con los datos (user id, cuando se mando, etc.)

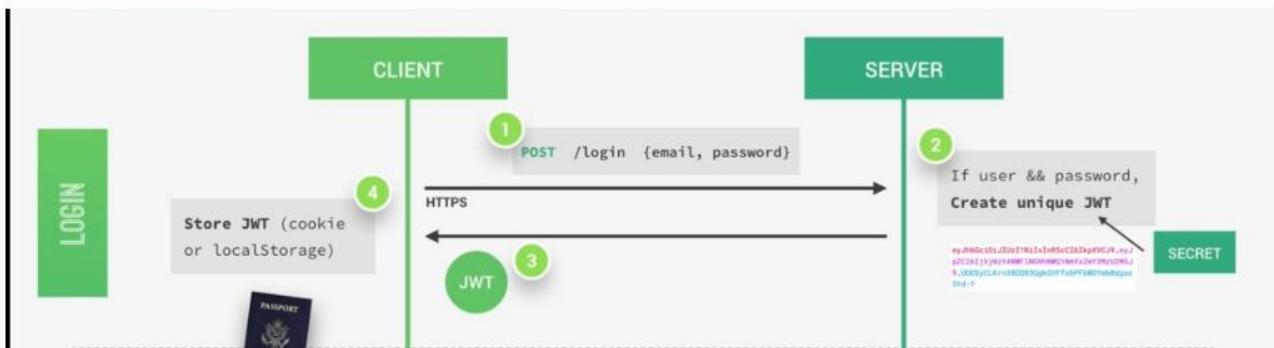
Las credenciales del usuario viajan solo 1 vez. Además, el token **NO** se almacena del lado del servidor para validar.

El uso de JWT incrementa la eficiencia en las aplicaciones evitando hacer múltiples llamadas a la base de datos.

- user: fiuba / pass: k@X4R\$KFEbCn
- Authorization:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1aWQiOjEyMzQ1Njc4OTAsIm5hbWUiOiJKb2hulERvZSIsInByb2YiOiJPV05FUilslm9yZyl6ODM0NzUyM30.2PWhCiy6sgDeBhGFbC1Ws1wloGgy7eY-44uey_aR0eo
- Authorization - Descriptado: <https://jwt.io/#debugger-io>

HEADER: ALGORITHM & TOKEN TYPE	PAYLOAD: DATA	VERIFY SIGNATURE
{ "alg": "HS256", "typ": "JWT" }	{ "uid": 1234567890, "name": "John Doe", "prof": "OWNER", "org": 8347523 }	HMACSHA256 (base64UrlEncode(header) + "." + base64UrlEncode(payload), hQo245thgSD\$KFJtg#%ebvjqf?2f3idJjdvl)

Los JWT pueden ser mensajes solo firmados, solo encriptados, o ambos. Si un token es solo firmado pero no encriptado, cualquiera puede leer su contenido, pero si no se conoce la clave privada, no puede modificarlo, por que al validar la firma no coincidiría.



El Server valida la firma de JWT para saber que lo que él envió al cliente no se modificó, y utiliza la información del mismo.

De esta forma se sigue siendo stateless, y generalmente se hace más eficiente la validación del usuario, sin tener que acceder a un medio persistente a validar si el token es válido y a quién pertenece el mismo.



Refresh Tokens

Los Access token deberían tener un tiempo limitado de vida, por lo que aparecen los **refresh token**.

Este es otro token que sirve para **un solo uso** y es utilizado para obtener un nuevo Access token. Es una credencial que permite obtener nuevos tokens sin necesidad de usar las credenciales de usuario y password nuevamente.

La aplicación misma se autologgea transparentemente.

Versionado

Rest no provee un mecanismo definido para versionado, pero se suelen ver estas estrategias:

- Usando la URI:

```
https://api.fi.uba.ar/v1  
https://apiv1.fi.uba.ar  
https://api.fi.uba.ar/20211101/
```

- Usando un Custom Header:

```
Accept-version: v1  
Accept-version: v2
```

- Usando el Header Accept:

```
Accept: application/vnd.example.v1+json  
Accept: application/vnd.example+json;version=1.0
```

Si hago un cambio a uno de los endpoints, la API que mis usuarios utilizaban por ahí dejan de andar. Por ahí tengo que armar nuevas versiones.

Hateoas

```
{  
    pageNumber: 1,  
    totalItemCount: 25,  
    pageItemCount: 10,  
    _links: {  
        * next: {  
            href: http://localhost:8080/cursos?pagina=2,  
            type: "application/vnd.cloud.programar.hateoas.Page"  
        },  
        * previous: {  
            href: http://localhost:8080/cursos?pagina=0,  
            type: "application/vnd.cloud.programar.hateoas.Page"  
        },  
        * self: {  
            href: http://localhost:8080/cursos?pagina=1,  
            type: "application/vnd.cloud.programar.hateoas.Page"  
        }  
    },  
    _embedded: {  
        items: [  
            {  
                codigo: "cod-10",  
                titulo: "Curso número 10",  
                unidadesDidacticasCompletadas: 2000,  
                _links: {  
                    * self: {  
                        href: http://localhost:8080/cursos/cod-10,  
                        type:  
                    }  
                }  
            }  
        ]  
    }  
}
```

Respuestas de los Endpoints

- Mantener lo más estandarizadas a las mismas
- Reducir el tamaño de la respuesta a lo necesario
- Utilizar código de errores http

Ejemplos:

```
HTTP CODE: 401 {
    "success": false, // solo informativo, el error se define por el HTTP CODE
    "message": "Invalid email or password",
    "error_code": 1308,
    "data": {}
}

HTTP CODE: 200 {
    "success": true,
    "message": "User logged in successfully", // optional in success responses
    "data": { }
}
```

Donde data es un objeto que contiene un mapa de otros objetos

El campo de data se usa para pasarle info al usuario.

```
{
    "success": true,
    "message": "User found",
    "data": [
        {
            "user": {
                "id": 2,
                "name": "Juan",
                "email": "juan@fi.uba.ar",
                "city": {
                    "id": 3,
                    "name": "Buenos Aires",
                    "country": {
                        "id": 2,
                        "name": "Argentina",
                        "code_country": "AR",
                        "avatar": " //localhost:3000/api/v1/country AR.png ",
                    }
                }
            },
            "role": "client",
            "favorites": ["blue", "red", "white"]
        }
    ]
}
```

Paginado, Filtros y Ordenamientos

HTTP CODE: 200

```
{
    "success": true,
    "metadata" : {
        "page": 5,
        "per_page": 20,
        "page_count": 20,
        "total_count": 521,
    }
    "data" : {....}
}
```

usando headers

```
HTTP/1.1 200
Pagination-Count: 100
Pagination-Page: 5
Pagination-Limit: 20
Content-Type: application/json
```

Filtros y ordenamiento suele usarse como parámetros del querystring o del body

Logging, Health, Metrics

Tener logs, métricas y puntos de control ayudan a detectar los problemas antes de que realmente llegue. Se suelen agregar endpoints para verificar o monitorear que la API esté viva y obtener datos de uso de memoria, etc.

- /health
- /metrics

Ejemplo de Endpoint de una API REST

Ejemplo de Login:

Descripción: Permite que los usuarios inicien sesión.

Request:

- Headers: **Authorization: Basic {base64(email:password)}**
- **POST /api/v1/users/token**

Response:

Código HTTP: 200 OK

```
{  
    "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",  
    "refresh_token": "m9obiBEb2UiLCJpYXQiOjE1MTYyMzkwMj..."  
}
```

Código HTTP: 401 Unauthorized si las credenciales son incorrectas.

Tipos de parametros

1. URI Path Parameter ({movie_name}):

- Forma parte de la estructura principal de la URL y representa un recurso específico.
- En este caso, {movie_name} es el identificador principal de la película que deseas calificar.
- Ejemplo:
`/api/v1/movies/Harry%20Potter%20and%20the%20Prisoner%20of%20Azkaban/ratings`

2. Query Parameter (director):

- Es un parámetro opcional que se agrega a la URL después de un signo de interrogación ? y proporciona información adicional para refinar la búsqueda o acción.
- En este caso, director ayuda a especificar aún más la película cuando hay múltiples películas con el mismo movie_name.
- Ejemplo completo:

```
/api/v1/movies/Harry%20Potter%20and%20the%20Prisoner%20of%20Azkaban/ratings?director=Alfonso%20Cuarón
```

Entonces: el path parameter define el recurso principal, mientras que el query parameter es un filtro que permite especificar más la búsqueda para evitar ambigüedades cuando puede haber elementos con el mismo nombre

Diferencias Clave:

Característica	Path Parameter ({movie_name})	Query Parameter (director)
Posición	Parte del "camino" principal de la URL	Parte después de <code>?</code> en la URL
Uso	Identifica recursos específicos	Filtro o información adicional
Requerido	Generalmente obligatorio	Opcional, depende del diseño
Finalidad	Define la ruta del recurso	Refina la solicitud

Request Body:

- **Ubicación:** El body es el contenido de la solicitud HTTP y se envía como parte del mensaje, no en la URL.
- **Formato:** Puede estar en diferentes formatos como JSON, XML, o form-data. Ejemplo de un body en JSON:

```
{
    "rating": 9,
    "comment": "Great movie!"
}
```

- **Uso típico:** Se usa para enviar datos complejos y estructurados al servidor, como:
 - Crear o actualizar un recurso (por ejemplo, enviar detalles de una nueva película o calificación).
 - Enviar formularios o datos de usuario.
- **Visibilidad:** El body no es visible en la URL y generalmente no tiene restricciones de longitud como los query parameters.

Diferencias Clave:

Característica	Query Parameter	Request Body
Ubicación	Parte de la URL después de <code>?</code>	Contenido de la solicitud HTTP
Propósito	Filtrar, buscar, o enviar datos simples	Enviar datos complejos o grandes
Formato	Clave-valor (<code>key=value</code>) en la URL	JSON, XML, <code>form-data</code> , etc.
Visibilidad	Visible en la URL	No visible en la URL
Longitud	Limitado (generalmente < 2048 caracteres)	Sin limitación significativa

El query parameter sirve para filtrar o identificar recursos, mientras que el body se usa para enviar datos que afectan al recurso.

Otro ejemplo con parámetros:

Descripción: Buscar las películas calificadas por un usuario

Request:

- Headers: Authorization: Bearer {access_token}
- GET /api/v1/users/{username}/rated_movies

Se asume que no puede haber dos usuarios con el mismo nombre de usuario.

Response

Código HTTP: 200 OK

```
{  
    "rated_movies": [  
        {  
            "movie_name": "IT",  
            "movie_director": "Andrés Muschietti",  
            "movie_rating": 8  
        },  
        {  
            "movie_name": "Hocus Pocus",  
            "movie_director": "Kenny Ortega",  
            "movie_rating": 10  
        }  
    ]  
}
```

Código HTTP: 401 Unauthorized si el token es invalido o esta expirado

Código HTTP 404 Not Found si el usuario buscado no se encuentra.

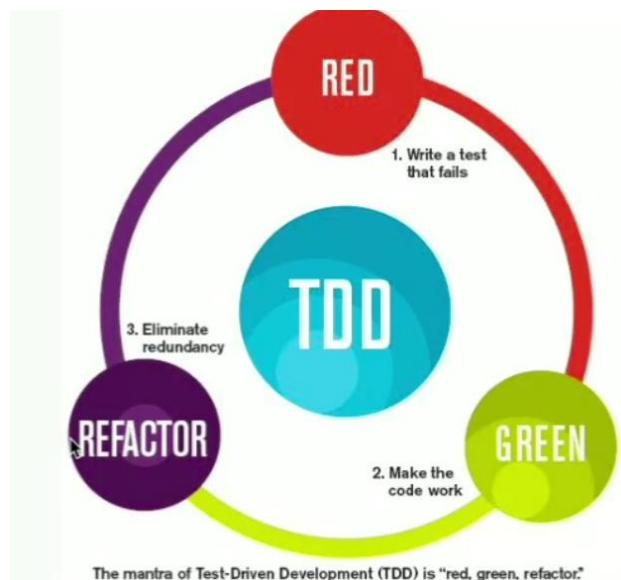
Refactoring

KATA / CODING DOJO

Kata: ejercicios de programación que uno ejecuta en un lugar sin riesgo, como un Dojo, donde no estoy en producción. Con las prácticas de programación puedo, por ejemplo, probar un framework nuevo sin necesidad a esperar a un próximo proyecto, sino ir practicando.

El objetivo no es llegar a una respuesta correcta, sino aprender; practicar no solucionar.

TDD: Test Driven Developement



1. No escribas código si no hay un test que falla
2. No agregues más de un test; se trabaja de a un test fallando
3. No escribas más código del que hace pasar tu test.

Refactoring

Una técnica para reestructurar una estructura de código existente, alterando la estructura interna sin cambiar su comportamiento externo.

Para hacer refactoring es importante contar una base amplia de tests unitarios, para asegurarnos que ese comportamiento externo no cambio, y luego aplicando los refactoreos propuestos.

Pasos del refactoring:

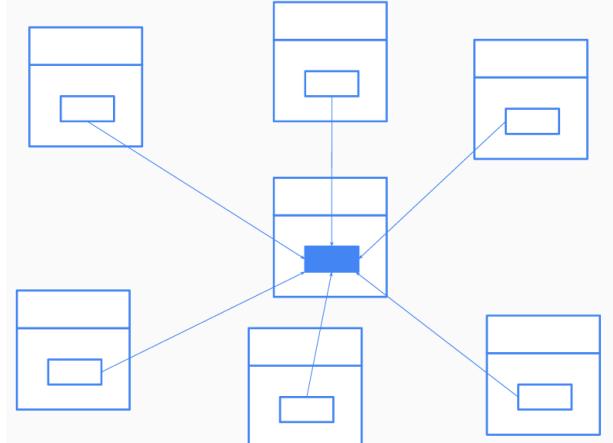
1. Asegurarse que todos los tests pasen
2. Encontrar un code smells (código duplicado, métodos muy largos, clases largas con muchos métodos, tener muchos parámetros en una función, switch statements, data class (todo set y gets))
3. Encontrar el refactoring
4. Aplicar refactoring

Code Smells

Es código que tiene características que indican que puede haber un problema/mala práctica sobre cómo se implementó ese código. A simple vista puedo no haber bugs, pero un code smells puede estar, por ejemplo, violando ciertas buenas prácticas de diseño que pueden llevar a tener una mala calidad de código y poor performance.

Ejemplos:

- Código duplicado
- Métodos muy extensos
- Clases muy extensas (que hacen muchas cosas)
- Funciones con muchos parámetros
- Divergent change: Cuando una clase tiene múltiples motivos por los cuales cambiar, por lo que tiene múltiples responsabilidades
- Shotgun Surgery: cuando cambiar código en un lugar requiere un cambio desencadenado en muchos (falta de encapsulación).



- Feature envy: un método o función de una clase utiliza excesivamente los métodos o datos de otra clase en lugar de los propios.
- Data Clumps: tenemos muchos métodos que reciben un conjunto de parámetros diferentes, que están desacoplados pero si vemos que muchas funciones llaman al mismo conjunto de parámetros podríamos intuir que todos se relacionan bajo una misma clase.

```
method1(■ ▲ ♦ ○)
method2(■ ▲ ♦ ○)
method3(■ ▲ ♦ ○)
method4(■ ▲ ♦ ○)
```

- Primitive obsession:

```
double money;
String phone;
String zipCode;
String password;
```

- Switch statements
- Lazy Class: una clase que no realiza suficiente trabajo útil. Puede ser una clase vacía, o una que solo contiene métodos o funcionalidades mínimas que podrían ser fácilmente absorbidas por otras clases o eliminadas por completo.
- Message chains: encadenar muchos outputs de funciones como inputs de muchas funciones y que la cadena se extienda demasiado:

■ . ■ () . ■ () . ■ () . ■ ()

- Data class: una clase que solo tiene métodos para acceder a sus atributos/actualizarlos sin tener una funcionalidad particular:

Cuenta
Código
Persona
Categoría
Rubro
contactos
getCodigo()
getPersona()
setPersona()
getCategoria()
setCategoria()
getRubro()
setRubro()
getContactos()
setContactos()