ARQUITECTURA DE SOFTWARE

Iker Caballero Bragagnini

Tabla de contenido

El diseño y la arquitectura

- Desde siempre ha habido confusión entre los términos de diseño y arquitectura y las similitudes y diferencias entre ellos, por lo que es necesario dar una definición para establecer un entendimiento común de ambos conceptos

    - La palabra “arquitectura” se suele usar en el contexto de algo a un alto nivel que está separado de los detalles de bajo nivel, mientras que la palabra “diseño” suele implicar estructuras y decisiones a un bajo nivel. No obstante, esto no tiene mucho sentido cuando se ve lo que un arquitecto hace de verdad

        - Un arquitecto también tiene en cuenta los detalles al más bajo nivel que apoyan todas las decisiones hechas a alto nivel, siendo ambos tipos de decisiones (a bajo y a alto nivel) necesarias para el diseño

        - Lo mismo ocurre con el diseño de *software*, los detalles a bajo nivel y a alto nivel son parte del mismo todo: forman parte de un tejido continuo que define la forma del sistema

    - El objetivo de la arquitectura o diseño de *software* es minimizar los recursos humanos necesarios requeridos para construir y mantener un sistema

        - La medida de la calidad de sueño es simplemente la medida del esfuerzo requerido para satisfacer las necesidades del cliente. Si el esfuerzo es bajo, entonces el diseño es bueno, mientras que, si crece para cada nueva entrega, es malo

    - La mejor opción para el desarrollo organizativo es reconocer y evitar la confianza excesiva y comenzar a tener más en cuenta la calidad de la arquitectura de *software*

        - Para ello, es necesario conocer que es una buena arquitectura de *software*: es un diseño que minimiza el esfuerzo y maximiza la productividad

        - Los atributos que se necesitan conocer para poder llegar a ello son aquellos que se discuten en el desarrollo teórico subsiguiente

- Los sistemas de *software* proporcionan dos tipos de valor a los usuarios: comportamiento y estructura. Los programadores se tienen que encargar que ambos estén altos, pero uno se suele enfocar más en uno que en otro o en ninguno

- El primer valor tiene que ver con el comportamiento: los programadores se contratan para hacer que las máquinas se comporten de una manera determinada que haga que se pueda obtener o se ahorre dinero y tiempo. Esto se hace a través de desarrollar una especificación funcional o requerimientos, los cuales se traducen en código

    - Cuando la máquina viola estos requerimientos, los programadores intentan *debuggear* o arreglar los problemas. No obstante, esto no es todo lo necesario

- El segundo valor tiene relación con el *software* y su arquitectura. Este *software* debe ser "suave", de modo que sea fácil cambiarlo de manera proporcional al alcance del cambio y no a su forma

    - La diferencia entre el alcance y la forma es lo que hace que incrementen los costes de desarrollo de *software*. Cada petición de cambio es más difícil de encajar en el sistema debido a que la forma del sistema puede no coincidir con la forma del cambio solicitado

    - El problema, claramente, está en la arquitectura del sistema: cuando más se prefiera una forma a otra, más probable es que las nuevas características sean más difíciles de encajar en la estructura. Por lo tanto, las arquitecturas deberían ser agnósticas a la forma del cambio para ser prácticas

- Las cosas urgentes normalmente son de gran importancia, mientras que las cosas no tan importantes no suelen ser urgentes

    - El primer valor del *software*, el comportamiento, es urgente pero no siempre es particularmente importante, mientras que el segundo valor del *software*, la arquitectura, es importante pero no siempre es urgente

        - Los problemas más comunes son elevar cosas no importantes y urgentes al estado de cosas importantes y urgentes, ignorando así la importancia de la arquitectura

Los paradigmas

- Los tres paradigmas principales que se desarrollarán son la programación estructurada, la programación orientada a objetos y la programación funcional
    - El primer paradigma que se adopta es el de la programación estructurada, el cual impone disciplina en la transferencia directa de control
        - Este paradigma se propuso por Dijkstra, el cual demostró que los saltos no restringidos (los comandos $goto$, que permiten hacer un salto desde cualquier lugar a cualquier lugar de una función) son dañinos par la estructura del programa
        - Una manera de remplazar los comandos de este tipo fue a través de comandos como if, $then$ o $else$ y do, $while$ o $until$
    - El segundo paradigma que se adopta es el de la programación orientada a objetos, el cual impone disciplina en la transferencia indirecta de control
        - Dahl y Nygaard descubrieron que se podía crear una clase que permitiera construir los objetos pertenecientes a esta a través de una función constructora, que las variables locales se podían usar como variables de instancia y que las funciones anidadas se podían convertir en métodos de estos objetos
        - Esto llevó al descubrimiento del polimorfismo a través del uso disciplinado de punteros a funciones
    - El tercer paradigma, adoptado muy recientemente, es el de la programación funcional, el cual impone disciplina en la asignación
        - Este paradigma es resultado de la investigación de Church, el cual inventó el cálculo lambda, el cual tiene como concepto fundamental la inmutabilidad (el valor de los símbolos no cambia). Esto significa que un lenguaje funcional no tiene una manera de asignar valores a las funciones
        - Existen maneras de cambiar el valor de las variables, pero bajo condiciones estrictas

- Los tres paradigmas quitan capacidades al programador, imponiendo algún tipo de disciplina extra que es negativa en su intención. Los paradigmas, por tanto, dicen lo que no hay que hacer

    - Los tres paradigmas juntos quitan la posibilidad de usar comandos $goto$, punteros de funciones y asignación de valores

- La programación estructurada contiene varios aspectos importantes que se tienen que estudiar detalladamente para poder utilizar este paradigma

    - Dijkstra reconoció que la programación es un problema difícil, de modo que se puede aplicar la disciplina matemática de la demostración o *proof*: las estructuras demostradas pueden usarse por los programadores para unirse con otras con código para que estas sean correctas

        - Durante su investigación Dijkstra descubrió que ciertos usos de los comandos $goto$ prevenían que los módulos se pudieran descomponer recursivamente en unidades menores para poder realizar demostraciones de algoritmos (a través de un enfoque de dividir para vencer)

        - No obstante, Dijkstra reconoció que los buenos usos de comandos $goto$ correspondían a estructuras de control de selección e iteración simples como if, $then$, $else$, do y $while$. Los módulos que solo usaban estos podían ser recursivamente subdivididos en unidades demostrables

    - Las estructuras de control combinadas con ejecuciones secuenciales eran especiales, dado que se había demostrado que todos los programas se pueden construir a través de 3 estructuras básicas: la secuencia, la selección y la iteración

        - Esto fue muy importante porque las estructuras de control mínimas que hacían que un módulo fuera demostrable eran las mismas estructuras de control mínimas con las que se podía construir cualquier programa, haciendo posible el nacimiento de la programación estructurada

        - Dijkstra demostró que los comandos secuenciales se podían demostrar a través de enumeración simple. Esta técnica trazaba matemáticamente los insumos de una secuencia a los resultados de una secuencia (como en una demostración normal)

        - También demostró la selección a través de la re-aplicación de la enumeración: cada camino a través de la selección se enumeró, y si ambos caminos producían eventualmente resultados matemáticos apropiados, la demostración era sólida

        - Para demostrar la iteración, Dijkstra usó la inducción matemática: demostrando el caso base para 1, demostró que si para N era correcta, para $N + 1$ también lo es (a través de enumeración). También demostró que los criterios de comienzo y de final de la iteración eran correctos a través de la enumeración

- La programación estructurada permite que los módulos se descompongan recursivamente en unidades demostrables, que significa que los módulos se pueden descomponer funcionalmente

    - Se puede tomar un problema de una gran escala y descomponerlo en funciones de alto nivel, permitiendo a la vez la descomposición de cada una de las funciones en funciones de menor nivel hasta el infinito. Además, cada una de estas funciones se puede representar usando las estructuras de control restringido de la programación estructurada

- Aunque al final la jerarquía euclidiana de teoremas para la programación nunca sucedió (la idea de Dijkstra), otra manera de poder probar que los códigos son correctos es a través del método científico

    - Las teorías científicas no se pueden demostrar como en matemáticas, sino que se toman medidas de manera repetida y se muestra que concuerda con las hipótesis. La naturaleza de las teorías científicas es que son falsificables, pero no demostrables

    - La ciencia funciona a través de demostrar que las teorías son incorrectas, más que correctas. De este modo, la ciencia es la disciplina que se basa en demostrar que una proposición es falsa, mientras que las matemáticas es la disciplina que se basa en demostrar que una proposición es cierta

- El *testing* o la comprobación muestra la presencia de *bugs*, pero no su ausencia, de modo que se puede demostrar que un programa es incorrecto con un *test*, pero no se puede demostrar que es correcto. Todo lo que los *tests* pueden hacer es demostrar que un código es lo suficientemente correcto para el propósito de uno

    - Las implicaciones de esto es que el desarrollo de *software* no es algo matemático, aunque se manipulan estructuras matemáticas, sino que es más como una ciencia: se demuestra lo correcto a través de no poder demostrar la equivocación

    - Tales demostraciones de equivocaciones solo se pueden aplicar a programas demostrables. Un programa que no es demostrable (por ejemplo, al utilizar comandos $goto$) no puede ser

considerado correcto, independientemente de cuantos *tests* se realicen

- La programación estructurada fuerza a descomponer recursivamente un programa en un conjunto de funciones demostrables pequeñas, posteriormente usando *tests* para demostrar que esas funciones son incorrectas. Si no se puede demostrar eso, se considera que las funciones son lo suficientemente correctas

- La base de una buena arquitectura es el entendimiento y la aplicación de los principios del diseño orientado a objetos (OO), de modo que es necesario analizar diversos conceptos clave de este paradigma
  - Uno de los tres conceptos que definen la programación OO es la encapsulación, dado que los lenguajes OO proporcionan encapsulación sencilla de datos y funciones
    - Como resultado, una línea puede dibujarse alrededor de un conjunto de datos y funciones cohesivo. Fuera de esa línea, los datos se esconden y solo se conocen algunas funciones
    - Este concepto se entiende sobre todo a través de datos miembros privados y las funciones miembro públicas de una clase
  - Esta idea no es única de un lenguaje de programación orientado a objetos, sino que también se tenía encapsulación perfecta en C
    - Los usuarios de $point.h$ no tienen acceso a miembros de $struct\ Point$: pueden llamar a la función $makePoint()$ y $distance()$, pero no pueden acceder a la implementación de $Point$ o sus funciones. Esto es encapsulación perfecta en C, y suele consistir en declarar estructuras de datos y funciones en archivos de cabecera o *header files*, e implementarlas en los archivos de implementación

point.h

```
struct Point;
struct Point* makePoint(double x, double y);
double distance (struct Point *p1, struct Point *p2);
```

point.c

```
#include "point.h"
#include <stdlib.h>
#include <math.h>

struct Point {
double x,y;
};
```

```
struct Point* makepoint(double x, double y) {
struct Point* p = malloc(sizeof(struct Point));
p->x = x;
p->y = y;
return p;
}

double distance(struct Point* p1, struct Point* p2) {
double dx = p1->x - p2->x;
double dy = p1->y - p2->y;
return sqrt(dx*dx+dy*dy);
}
```

- Cuando salió C++, un lenguaje de programación OO, la encapsulación perfecta se rompió debido a que el compilador de C++ necesitaba que las variables miembros se especificaran en el archivo de cabecera de la clase, haciendo que sean visibles. No obstante, esta se pudo arreglar parcialmente a través de la introducción de palabras como $public$, $private$ y $protected$

point.h

```
class Point {
public:
Point(double x, double y);
double distance(const Point& p) const;

private:
double x;
double y;
};
```

point.cc

```
#include "point.h"
#include <math.h>

Point::Point(double x, double y)
: x(x), y(y)
{}

double Point::distance(const Point& p) const {
double dx = x-p.x;
double dy = y-p.y;
return sqrt(dx*dx + dy*dy);
}
```

- Java y C# simplemente abolieron la separación entre archivos de cabecera e implementación, de modo que es imposible separar la declaración de la definición de una clase. Por estas razones, es difícil aceptar que los lenguajes de OO tienen poca encapsulación

- La herencia es otro de los aspectos claves de los lenguajes OO, y se basa en la re-declaración de un grupo de variables y funciones dentro de un alcance cerrado

  - En el ejemplo, la estructura de datos $NamedPoint$ actúa como un derivado de $Point$, lo cual se debe a que el orden de los primeros dos campos de $NamedPoint$ son los mismos que en $Point$. En este caso, $NamedPoint$ es un superconjunto de $Point$ y mantiene la ordenación de los miembros que corresponde a $Point$

namedPoint.h

```
struct NamedPoint;

struct NamedPoint* makeNamedPoint(double x, double y, char* name);
void setName(struct NamedPoint* np, char* name);
char* getName(struct NamedPoint* np);
```

namedPoint.c

```
#include "namedPoint.h"
#include <stdlib.h>

struct NamedPoint {
double x,y;
char* name;
};

struct NamedPoint* makeNamedPoint(double x, double y, char* name) {
struct NamedPoint* p = malloc(sizeof(struct NamedPoint));
p->x = x;
p->y = y;
p->name = name;
return p;
}

void setName(struct NamedPoint* np, char* name) {
np->name = name;
}

char* getName(struct NamedPoint* np) {
return np->name;
}
```

main.c

```
#include "point.h"
#include "namedPoint.h"
#include <stdio.h>

int main(int ac, char** av) {
struct NamedPoint* origin = makeNamedPoint(0.0, 0.0, "origin");
struct NamedPoint* upperRight = makeNamedPoint (1.0, 1.0, "upperRight");
printf("distance=%f\n",
distance(
(struct Point*) origin,
(struct Point*) upperRight));
}
```

- Este truco se utilizaba mucho antes de la invención de los lenguajes OO, y se suele usar en C++ par implementar la herencia simple. No obstante, no se tenía herencia completa, sino que era singular y esta no era tan conveniente

- Los lenguajes OO permitieron hacer la máscara en estructuras de datos de manera significativamente más conveniente, sin la necesidad de hacer *casting* y pudiéndose aplicar en múltiples instancias

- El aspecto final más importante de los lenguajes es el polimorfismo, que se basa en la creación de una sola interfaz de entidades de diferentes tipos. Se considera una aplicación de punteros a funciones

  - Se tenía polimorfismo en C antes de los lenguajes OO. En el ejemplo, $getchar()$ se lee desde $STIDIN$ y $putchar()$ se escribe en $STDOUT$, pero no se sabe que dispositivos son esos, de modo

que la función es polinómica: el comportamiento depende del tipo de $STDIN$ y $STDOUT$

```
#include <stdio.h>

void copy() {
int c;
while ((c=getchar()) != EOF)
putchar(c);
}
```

- El sistema operativo de UNIX requiere que cualquier *driver* del dispositivo IO proporcione cinco funciones estándar: $open$, $close$, $read$, $write$ y $seek$. Las firmas de estas funciones deben ser idénticas para cualquier *driver* de IO

- Como ejemplo, la estructura de datos $FILE$ contiene cinco punteros a funciones, y el *driver* de IO para la consola definirá estas funciones y cargará la estructura de datos con sus direcciones. Si STDIN se define como $FILE^*$, y si apunta a la estructura de datos de la consola, entonces $getchar()$ se implementa de la siguiente manera:

```
struct FILE {
void (*open)(char* name, int mode);
void (*close)();
int (*read)();
void (*write)(char);
void (*seek)(long index, int mode);
};


#include "file.h"

void open(char* name, int mode) {/*...*/}
void close() {/*...*/};
int read() {int c;/*...*/ return c;}
void write(char c) {/*...*/}
void seek(long index, int mode) {/*...*/}

struct FILE console = {open, close, read, write, seek};


extern struct FILE* STDIN;

int getchar() {
return STDIN->read();
}
```

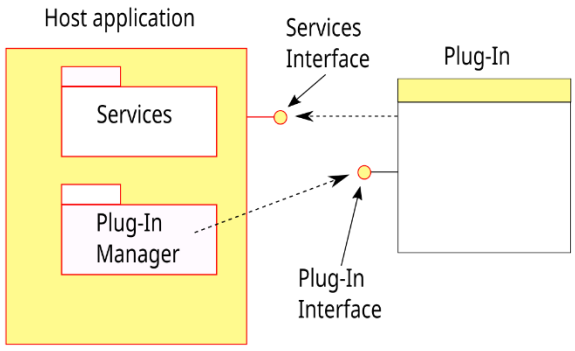
- De este modo, $getchar()$ simplemente llama a la función a la que se apunta por el puntero $read$ de la estructura $FILE$ que se apunta por STDIN. Por lo tanto, si se necesitara cambiar algo para poder leer el insumo, solo se necesita cambiar la función $read()$, pero no la función $getchar()$

- Los lenguajes OO no crearon el polimorfismo, pero lo hicieron más seguro y conveniente: el problema de usar punteros a funciones para crear comportamiento polimórfico es que son peligrosos, debido a que se necesita seguir unas convenciones

para inicializar, para llamar las funciones,etc. Los lenguajes OO eliminan estas convenciones, haciendo que el polimorfismo se vuelva trivial y que se imponga disciplina para la transferencia de control indirecta

- Lo mejor del polimorfismo es que no se necesitan hacer cambios en un programa cuando se introduce un nuevo dispositivo IO. Esto se puede entender mejor tomando como ejemplo el programa para la función $copy()$

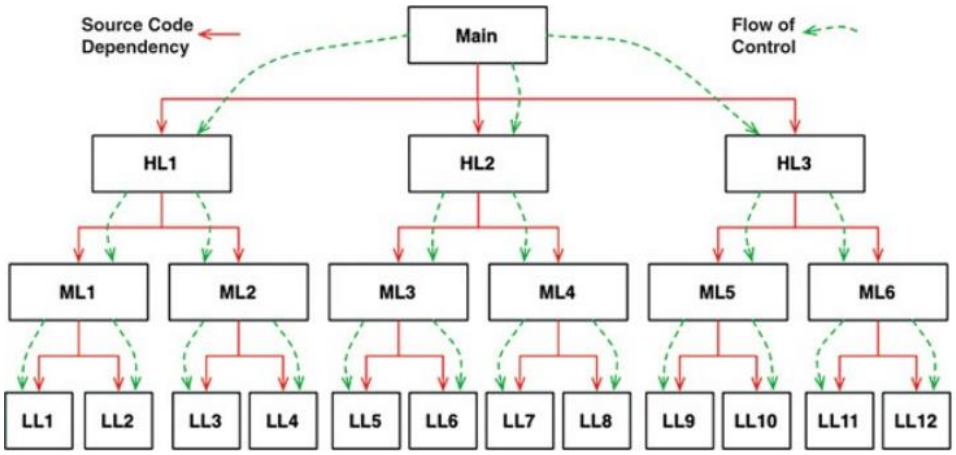
  - Esto se debe a que el código fuente del programa no dependería del código fuente de los *drivers* de IO: mientras que los *drivers* de IO implementen las 5 funciones estándar definidas en $FILE$, el programa siempre podrá usarlas (solo se necesita cambiar la definición de las funciones). Por lo tanto, los dispositivos IO se han convertido en *plugins* (componente de *software* que añade una característica a un programa existente)



  - Los programas deben ser independientes del dispositivo para que se puedan transferir y hacer las mismas cosas en otros dispositivos. La arquitectura de *plugin* se inventó para soportar este tipo de independencia entre dispositivos IO, y se ha implementado en casi todos los sistemas operativos

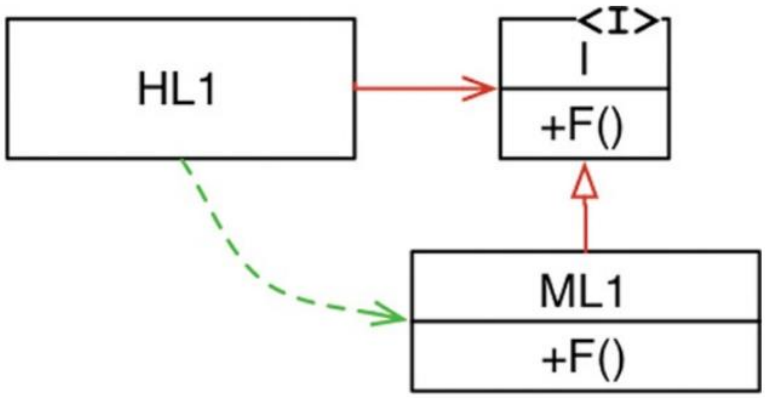
  - Los lenguajes OO permiten utilizar la arquitectura *plugin* en cualquier lugar y para cualquier tarea

- Antes de la introducción de un mecanismo seguro y conveniente para el polimorfismo, en un árbol de llamadas o *calling tree* las funciones principales (*main functions*) llamaban a funciones de alto nivel, que llamaban a su vez a funciones de nivel medio, y así. Las dependencias del código fuente seguían el flujo de control inexorablemente



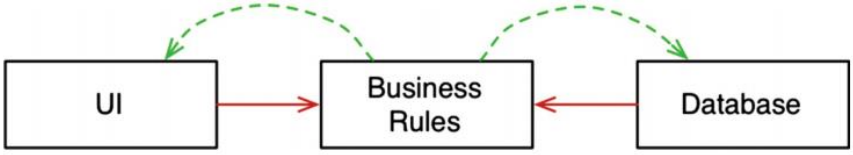
- Para que la función principal llame a una función de alto nivel, tenía que mencionar el nombre del módulo que contenía esta función (a través de $\#include$ en C), por lo que la función principal depende del módulo de alto nivel. De este modo, las dependencias del código fuente se dictaban por el flujo de control (orden en el que se ejecutan las instrucciones), pero las cosas cambian cuando el polimorfismo entra en juego

- Por ejemplo, en el esquema siguiente, el módulo HL1 llama a la función $F()$ en el módulo ML1 pero a través de una interfaz I (una invención del código fuente, no existe cuando se ejecuta, sino que solo se llama a la función desde el módulo). La relación de herencia entre ML1 y la interfaz I apunta en dirección contrario al flujo de control, lo cual se conoce como inversión de dependencia



- El hecho de que los lenguajes OO proporcionen polimorfismo seguro y conveniente hace que cualquier dependencia del código fuente se pueda invertir. A través de este enfoque los arquitectos de *software* pueden tener un control absoluto sobre la dirección de todas las dependencias de código fuente del sistema y no tiene por qué estar alineados con el flujo de control

- Para realizar esto, solo se necesitan crear tres tipos de archivo independientes: dos para los módulos y uno para la interfaz. No obstante, esto depende del lenguaje de programación y cómo es su flujo de control y dependencia del código fuente



- En muchas maneras, los conceptos de la programación funcional abarcan más que la programación en sí misma, y se basa principalmente en el cálculo lambda de Church

  - Para explicar lo que es la programación funcional, lo mejor es examinar ejemplos. Un ejemplo simple es imprimir el cuadrado de los primeros 25 enteros

    - En lenguaje Java (arriba) y Clojure (abajo), esto se implementaría de la siguiente manera:

```
public class Squint {
public static void main(String args[]) {
for (int i=0; i<25; i++)
System.out.println(i*i);
}
}
```

```
(println (take 25 (map (fn [x] (* x x)) (range))))
```

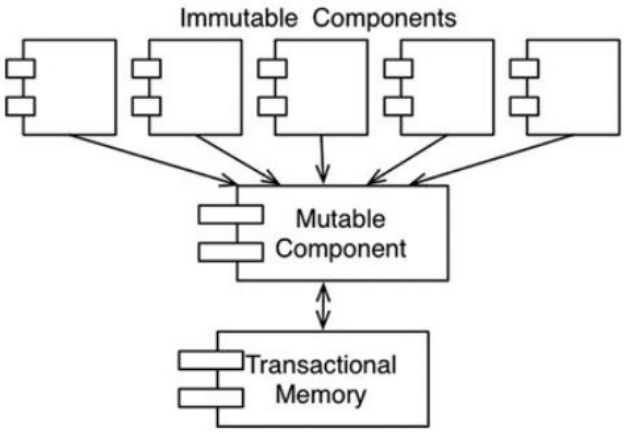
    - El programa de Java usa una variable mutable (una variable que cambia de estado durante la ejecución del programa), pero ese tipo de variable no existe en Clojure, dado que las variables se inicializan, pero nunca se modifican. Esto da un ejemplo perfecto de que las variables no varían en los lenguajes funcionales

  - Este último punto es importante, dado que todas las condiciones de carrera, *deadlocks* y problemas de actualización concurrente se deben a variables mutables. Todos los problemas que aparecen en aplicaciones concurrentes no pueden ocurrir si no hay variables mutables

    - Como arquitecto, uno tiene que estar muy interesado en los problemas de concurrencia: uno quiere estar seguro de que los sistemas que diseña serán robustos en la presencia de múltiples *threads* y procesadores. La pregunta clave es saber si es posible practicar la inmutabilidad

    - Si no se tiene almacenamiento y velocidad infinita del procesador, la inmutabilidad se puede ejercer, pero con ciertos compromisos

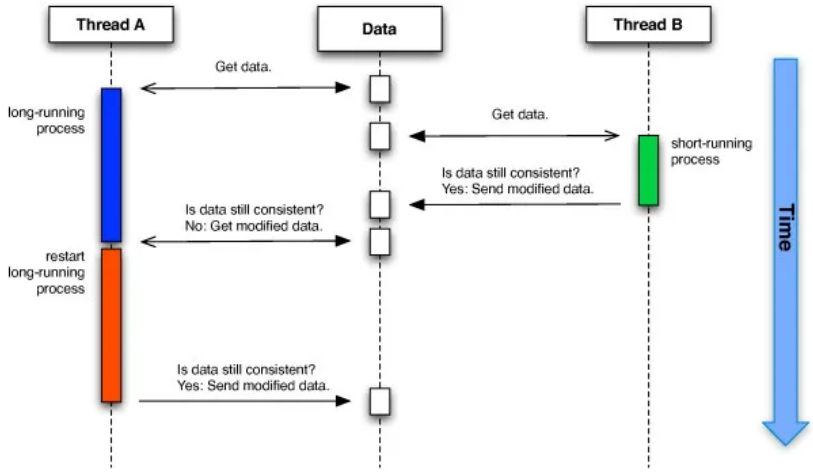
- Uno de los compromisos más comunes relacionados con la inmutabilidad es segregar la aplicación, o los servicios dentro de la aplicación, en componentes mutables e inmutables



  - Los componentes inmutables realizan sus tareas de manera puramente funcional, sin usar variables mutables. Los componentes de este tipo comunican con uno o más componentes que no son puramente funcionales, y permiten que el estado de las variables sea mutable

  - Debido a que el estado mutable expone estos componentes a todos los problemas de concurrencia, es común usar algún tipo de memoria transaccional para proteger las variables mutables de actualizaciones concurrentes y condiciones de carrera

  - La memoria transaccional simplemente trata las variables en memoria de la misma manera que una base de datos trata a los registros en un disco: protege a estas variables con un esquema de transacción



        - Los arquitectos deberían empujar la mayoría de procesamiento a componentes inmutables, y hacer que la cantidad de código en los componentes que permiten la mutabilidad sea mínima

        - Un servicio es una funcionalidad o un conjunto de funcionalidades de *software* (ejecución de una serie de operaciones, por ejemplo) con el objetivo de que pueda ser reutilizado por distintos clientes con distintos propósitos

    - Los límites de almacenamiento y velocidad se han incrementado exponencialmente en la actualidad, por lo que cada vez se necesita menos a los estados mutables

        - Como ejemplo de esto, se puede pensar en una aplicación bancaria que mantiene los balances de las cuentas de sus clientes, mutando estos balances con cada depósito y extracción. Si se guardaran, en cambio, todas las transacciones, se podrían sumar todas las transacciones de la cuenta para poder calcular los balances, sin necesitar variables mutables

        - Aunque esto solo parece plausible si se tuviera memoria y poder de procesamiento infinito para que funcione para siempre, puede ser que no se necesitara que funcione para siempre o que se tenga un poder de procesamiento y memoria suficiente para hacer que el esquema funcione por un tiempo razonable

        - Esta última es la idea detrás del *evento sourcing*, que es una estrategia en donde se guardan las transacciones, pero no el estado. De este modo, cuando el estado se requiere, simplemente se aplican todas las transacciones desde el comuenzo

        - Si se tuviera suficiente almacenamiento y poder de procesador, se podrían hacer las aplicaciones enteramente inmutables o funcionales

Los principios del diseño: los principios SOLID

- Un buen *software* se construye a través de un código limpio: si los bloques fundacionales están mal, la arquitectura no es importante, pero se puede crear un *software* caótico con buenos bloques. Por lo tanto, se suelen seguir unos cuantos principios conocidos como los principios SOLID

    - Los principios SOLID dicen como organizar las funciones y las estructuras de datos en clases, y cómo estas clases deberían estar interconectadas

        - La palabra “clase” no indica que estos principios solo se puedan aplicar a programación orientada a objetos. Una clase no es más que un grupo unido de funciones y datos, por lo que los principios aplican a todas estas agrupaciones

        - Existen cinco principios SOLID: el principio SRP, el principio OCP, el principio LSP, el principio ISP y el principio DIP

    - El objetivo de estos principios es la creación de estructuras *mid-level* de *software* que toleren cambios, sean fáciles de entender y que sean la base de componentes que se puedan usar en varios sistemas de *software*

        - El término “*mid-level*” se refiere al hecho de que estos principios se aplican por programadores que trabajan a nivel de módulos. Estos se aplican justo por encima del nivel del código y ayudan a definir tipos de estructuras de *software* usadas dentro de los módulos y componentes

        - Tal como es posible crear un caos con buenos bloques fundacionales, se puede crear un caos a nivel sistemático con buenos componentes *mid-level*. Por lo tanto, hay otros principios que aplican después de estos

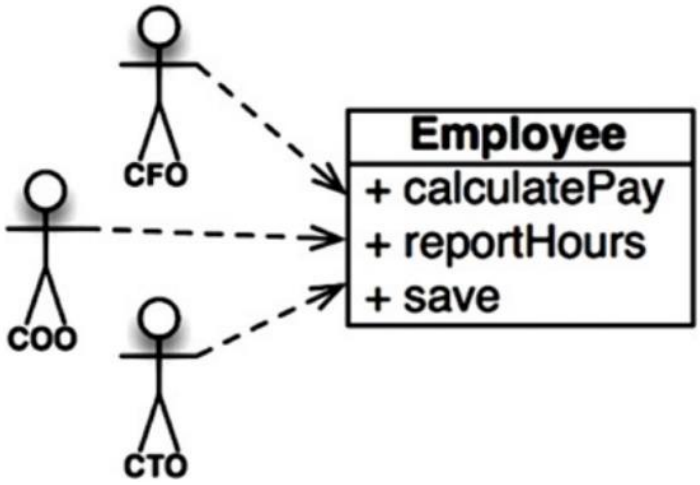
- El principio SRP (de *Single-Responsibility Principle*) se basa en que una buena estructura para un sistema de *software* se influencia mucho por la estructura social de la organización que usa, de modo que cada módulo debe tener solo una razón para cambiar

    - Un módulo solo debe ser responsable a un solo actor (grupo de usuarios o personas interesadas), que requiere el cambio

        - Con módulo uno se refiere a un archivo fuente o *source file*, pero como algunos lenguajes no usan archivos para contener el código, entonces se entiende un módulo como un conjunto cohesivo de funciones y estructuras de datos

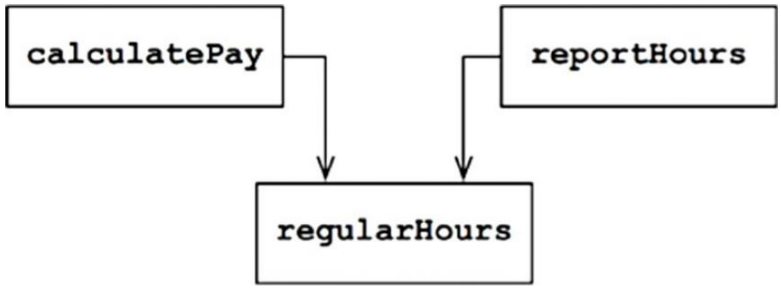
        - El principio se centra en esta cohesión, que es la fuerza que vincula el código responsable a un solo actor. No obstante, lo mejor para entender esto es ver las consecuencias de no seguir este principio

    - Como ejemplo, se considera una clase llamada $Employee$ para una aplicación salarial, la cual contiene tres métodos: $calculatePay()$, $reportHours()$ y $save()$



- Esta clase viola el SRP debido a que los tres métodos son responsables de tres actores muy diferentes: el método $calculatePay()$ se especifica por el CFO, el método $reportHours()$ se especifica por el COO y el método $save()$ por el CTO

- Al poner los tres métodos en una sola clase, se han juntado los diferentes actores entre ellos. Esto puede hacer que las acciones de unos actores afecten las actividades o elementos de otros cuando se afectan partes del código (dado que suele llamarse para otros métodos, por ejemplo) y que haya duplicación accidental

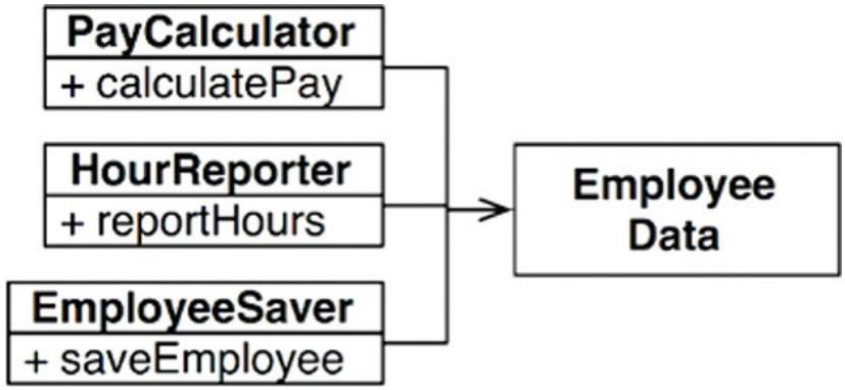


- Otro ejemplo de por qué se debe seguir este principio es el de que los cambios en el código se sobrepongan, que es normal en archivos fuente que contienen varios métodos y estructuras de datos

    - Si se junta código que es responsabilidad de diversos actores, estos pueden querer modificar la mima parte del código, de modo que la fusión de los cambios puede ser incompatible y generar problemas

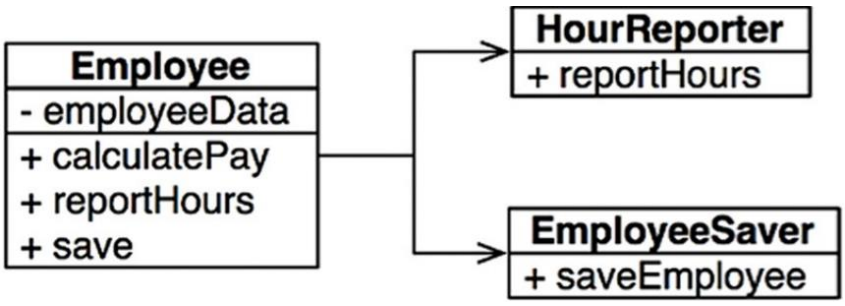
- Hay diferentes soluciones a este tipo de problema, pero todas pasan por separar los datos de las funciones

- La manera más obvia de solucionar este problema es a través de separar los datos de las funciones, de modo que se crean clases particulares con la porción de código necesario para desarrollar su función particular y se comparte el acceso a los mismos datos. De este modo, se evitan duplicaciones accidentales, pero ahora se tienen diversas clases en vez de una sola



- Una solución a este último problema es el uso de un patrón de fachada o *facade pattern*, de modo que se crea una sola clase (como en el ejemplo anterior) pero que instancia y delega las funciones a las clases separadas encargadas de cada función (llama a los objetos de las clases dentro de su misma estructura)

- Si se quiere mantener las funciones principales más cerca de los datos, se pueden mantener los métodos más importantes en la clase fachada (que se asume que tiene acceso a los datos) y llamar a los objetos de las otras clases encargadas de las funciones menos importantes



- Cada clase no tendrá solo una única función, dado que normalmente obtener el resultado deseado requerirá diferentes

funciones. Cada una de estas clases tendrá métodos privados (se limitará el alcance o *scope* para cada clase)

- El principio OCP (de *Open-Closed Principle*) se basa en la idea de que los sistemas de *software* sean fáciles de cambiar, tienen que diseñarse para permitir el comportamiento de estos sistemas para que cambien al añadir nuevo código, en vez de cambiar código existente

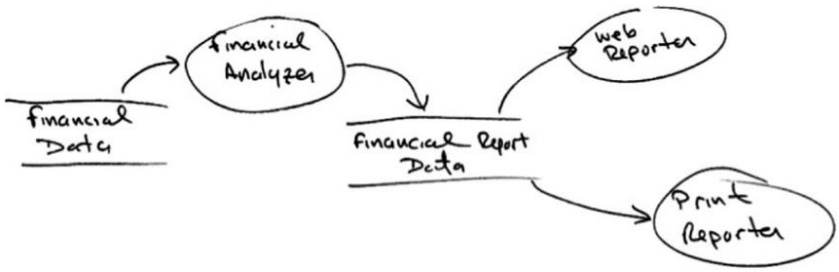
    - Un artefacto de *software* debería ser abierto para la extensión, pero cerrado para la modificación, de modo que el comportamiento del artefacto debe ser extensible, pero sin necesidad de modificarlo

        - Esta es la razón fundamental para el estudio de la arquitectura de *software*: si extensiones simples requieren cambios masivos en el sistema, este es un fracaso

        - Este principio es más importante cuando se considera el nivel de los componentes arquitectónicos

    - Para poder entender mejor esto se puede plantear un ejemplo simple: imaginando que se tiene un sistema que muestra un resumen financiero en una página web, donde los datos se pueden arrastrar y los números negativos están en rojo
        - Si se quisiera que estos mismos datos se reportaran como un informe en blanco y negro, se necesitaría hacer modificaciones en el código. Una buena arquitectura de *software* debería reducir la cantidad de código que se necesita modificar a casi cero

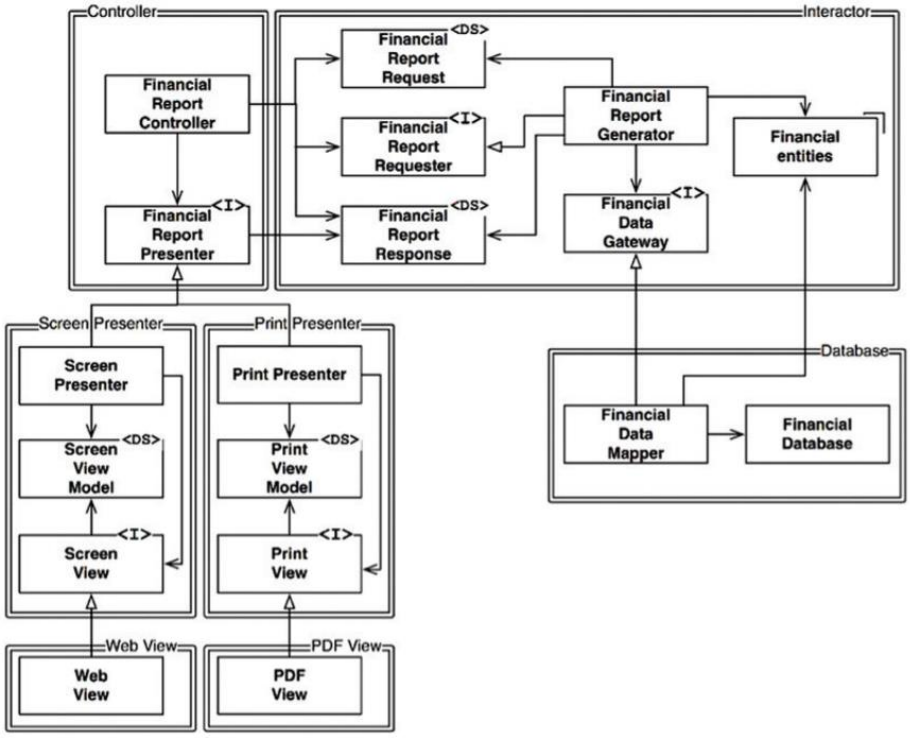
        - Esto último se logra a través de separar adecuadamente las cosas que cambian por diferentes razones (SPR) y organizando las dependencias entre estas cosas de manera adecuada (DIP). Aplicando el SPR, se puede realizar un esquema del flujo de datos que indica que generar el informe requiere de dos procesos: el cálculo de los datos a reportar y su presentación en formado web e impreso



        - Teniendo esta separación, se necesita organizar las dependencias del código fuente para asegurarse que los cambios en una de estas responsabilidades no afectan a la otra. Además, la nueva

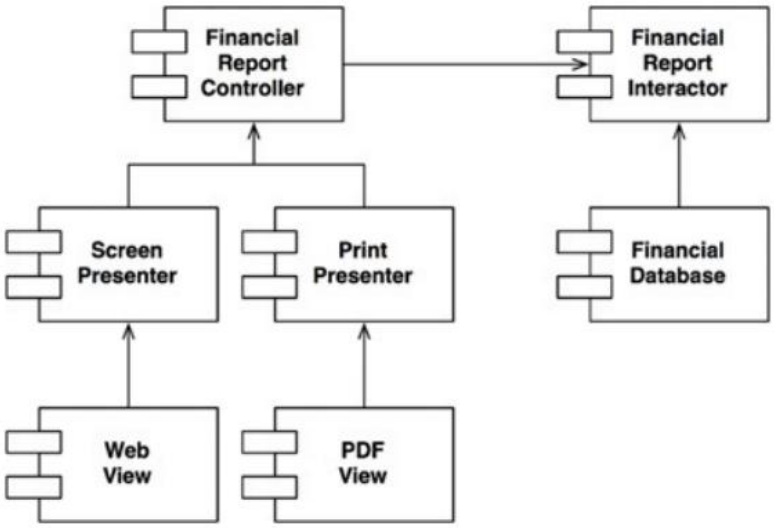
organización debe asegurarse que el comportamiento se puede extender sin necesidad de modificaciones posteriores

- Esto se hace al partir los procesos en clases y separando estas clases en componentes. A partir de ello, se puede realizar el siguiente esquema, en donde los componentes están delimitados por líneas dobles, las clases marcadas con “<I>” son interfaces y aquellas con “<DS>” son estructuras de datos



  - Todas las dependencias son dependencias del código fuente, de modo que una flecha que apunta de una clase A a una B quiere decir que la clase A menciona el nombre de la clase B (pero no al revés). Una flecha cerrada quiere decir que hay una relación de herencia, mientras que una flecha abierta quiere decir que se usan relaciones

  - Cada línea doble se cruza únicamente en una dirección, de modo que todas las relaciones de componentes son unidireccionales. Las flechas apuntan hacia los componentes que se quiere proteger del cambio en los componentes de dónde nace la flecha



- Esto crea un tipo de jerarquía de protección basada en la noción de niveles: los interactores son el concepto de mayor nivel, mientras que los otros componentes tienen un nivel menor (todos los otros componentes lidian con cuestiones periféricas)

- Esto es como el OCP funciona a nivel arquitectónico: los arquitectos separan la funcionalidad basándose en cómo, por qué y cuando cambia, y luego organizan esa funcionalidad separada en una jerarquía de componentes. Los componentes de mayor nivel se protegen de los cambios hechos en los componentes de menor nivel

  - Mucha de la complejidad de los diagramas anteriores se introdujo para asegurarse de que las dependencias entre componentes apuntaban a la dirección correcta (conocido como control direccional)

    - Algunas de las clases anteriores, como aquellas que son interfaces, se usan para invertir la dependencia que de otra manera existiría entre componentes

    - Otras interfaces usadas sirven para esconder información de otras clases que de otra manera exhibirían dependencias transitivas (dependencias indirectas), que violan el principio general de que las entidades de *software* no deberían depender de cosas que no usan directamente (ISP y CRP)

    - De este modo, aunque el objetivo sea proteger una clase A de cambios en una clase B, también se quiere proteger la clase B de cambios en A a través de esconder información interna de A

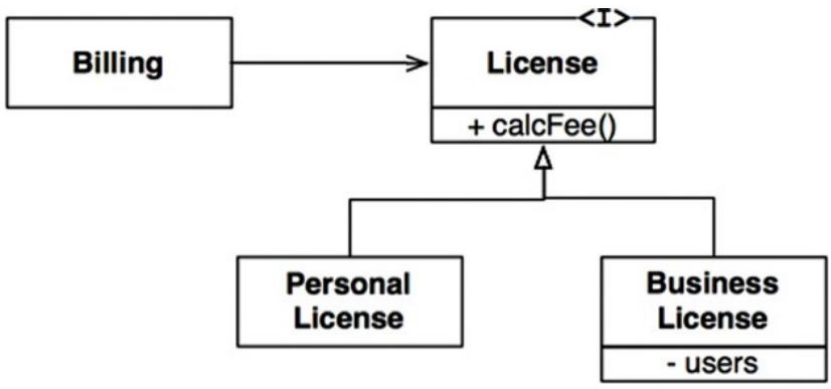
- El principio LSP (de *Liskov Substitution Principle*) es un principio que se basa en la idea de que, para construir sistemas de *software* a partir de partes

intercambiables, estas partes se deben adherir a un contrato que permita que estas partes sean intercambiables entre ellas

- Barbara Liskov proponía la necesidad de una propiedad sustitutiva del siguiente tipo: si para cada objeto $O1$ del tipo S hay un objeto $O2$ del tipo T tal que para todos los programas P definidos en términos de T, el comportamiento de P se intercambia cuando $O1$ se sustituye por $O2$, entonces S es un subtipo de T

    - Este es el principio LSP de Barbara Liskov, el cual se puede entender mejor a través de ejemplos

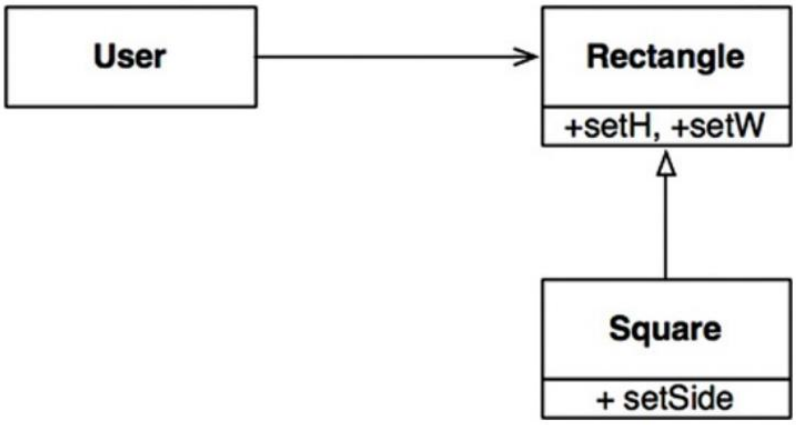
    - Como ejemplo, se imagina la existencia de la clase $License$, la cual tiene un método llamado $calcFee()$ y que se llama a través de la aplicación $Billing$. Hay dos subtipos de la clase $License$ dependiendo del algoritmo que utilizan para calcular la comisión por licencia: $PersonalLicense$ y $BusinessLicense$



    - Este diseño respeta el LSP debido a que el comportamiento de la aplicación $Billing$ no depende, de ninguna manera, en cuál de los dos subtipos de $License$ se utiliza. Ambos subtipos se pueden sustituir por el tipo $License$

- Un ejemplo famoso de la violación del principio LSP es el ejemplo del problema del rectángulo y el cuadrado

    - En este ejemplo, la clase $Square$ no es un subtipo propio de $Rectangle$, dado que la altura y el ancho de $Rectangle$ se pueden mutar de manera independiente, pero en $Square$ estos se deben mutar de manera conjunta o de la misma manera



- Como $User$ piensa que se está comunicando con la clase $Rectangle$, el código que puede fallar. En el código de ejemplo, si "…" fuera de la clase $Square$, la comprobación fallaría

```
Rectangle r = …

r.setW(5);

r.setH(2);

assert(r.area() == 10);
```

- La única manera de defenderse de contra una violación del LSP es añadir mecanismos para que $User$ detecte si $Rectangle$ es del subtipo $Square$ (como un comando if). Como el comportamiento de $User$ depende del tipo que se use, estos tipos no son sustituibles

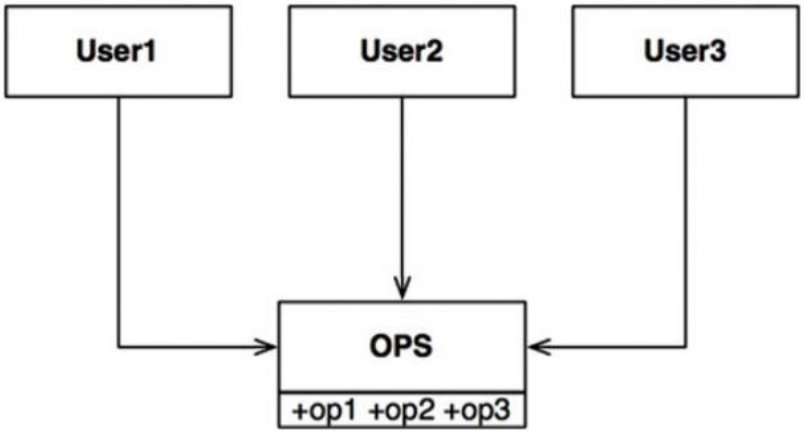
  - Aunque en los primeros años de la arquitectura de *software* el LSP se uso como una guía para la herencia de clases, ahora se ha convertido en un principio más general de diseño que pertenece a interfaces e implementaciones

    - Las interfaces en cuestión pueden ser de varias formas, pero el LSP siempre es aplicable porque hay usuarios que dependen de interfaces bien definidas, y en la sustituibilidad de las implementaciones de estas interfaces

    - Una violación de la sustituibilidad puede hacer que un sistema de *software* sea contaminado con una cantidad increíble de mecanismos extra

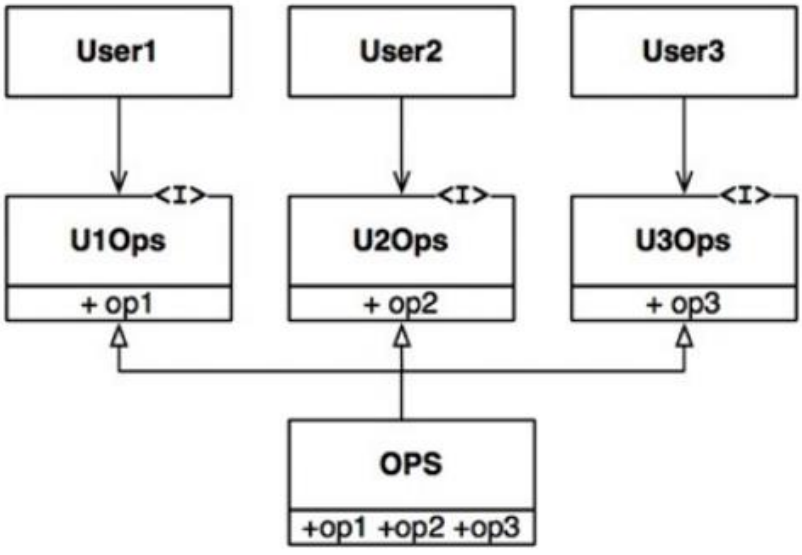
- El principio ISP (de *Interface Segregation Principle*) se basa en la idea de que los diseñadores de *software* deberían evitar depender de cosas que no se usan

- La mejor manera de entender el principio es a través de un ejemplo ilustrativo: en la situación del esquema, los usuarios usan las operaciones de la clase Ops, y se asume que cada uno de los usuarios solo usa una de las operaciones incluidas en las clases



  - Dependiendo del lenguaje en el que se implementa todo, claramente el código fuente de un usuario dependería de las otras operaciones que no utiliza. Esto significa que cambios en el código fuente de una operación de Ops fuerzan a usuarios que no la usen a recompilarse y redesplegarse, aunque no importe para el código de los usuarios

  - Este problema se puede resolver al segregar operaciones en interfaces, de modo que cada usuario dependa de una interfaz diferenciada que use las operaciones de Ops, pero no de Ops directamente, por lo que los cambios en partes de Ops no directamente usadas en código del usuario no afectarán



- Claramente, la descripción anterior depende críticamente del tipo de lenguaje de programación

        - Los lenguajes de programación estáticos como Java fuerzan a los programadores a crear declaraciones de usuarios de $import$, use o $include$. Son estas declaraciones $include$ en el código fuente que crean dependencias de código que fuerzan la recompilación y el redespliegue

        - En lenguajes de programación dinámicos como Python, estas declaraciones no existen en el código fuente, sino que se infieren en el tiempo de ejecución. Por lo tanto, no hay dependencias de código que fuerzan la recompilación y el redespliegue, y esa es la razón por lo que se suelen usar estos lenguajes para crear sistemas más flexibles y menos acoplados

    - Si uno da un paso atrás y mira las motivaciones raíz del ISP, en general, es malo depender de módulos que contienen más de lo que uno necesita. Esto es obviamente verdadero para las dependencias que pueden forzar la recompilación y el redespliegue (pero a nivel arquitectural)

        - Considerando, por ejemplo, un arquitecto trabajando en un sistema en el que necesita incluir un cierto marco F. Suponiendo que los autores de F lo han vinculado a una base D, entonces S depende de F y depende de D

        - Si se supone que D contiene características que F no usa, por lo que tampoco usa S, cambios en esas características de D pueden forzar el redespliegue de F y, en consecuencia, de S. En el peor caso, fallos en una de esas características en D pueden causar fallos en F y S

- El principio DIP (de *Dependence Inversion Principle*) se basa en la idea de que el código que implementa una política de alto nivel no debería depender del código que implementa detalles de bajo nivel, sino que los detalles deben depender de la política

    - El DIP nos dice que los sistemas más flexibles son aquellos en los que las dependencias del código fuente se refieren solo a abstracciones y no a concreciones

        - En lenguajes estáticos como Java, esto quiere decir que los comandos use , $import$ o $include$ solo deberían referirse a modelos fuente que contengan interfaces, clases abstractas u otros tipos de declaraciones abstractas (nada concreto).

        - Estas mismas reglas aplican para lenguajes dinámicos, solo que en estos casos es más difícil definir lo que es un módulo concreto: será un módulo en la que las funciones que se llaman se han implementado

        - Claramente, tratar esto como una regla no es realista porque los sistemas siempre dependen de concreciones: lo único que se debe procurar para confiar en estas concreciones (y no tener que abstraerlas) es que estas no cambien. Son las concreciones volátiles con las que se quiere evitar una dependencia (con cambios frecuentes y bajo desarrollo)

    - Cada cambio a una interfaz abstracta corresponde a un cambio de sus implementaciones concretas. Conversamente, los cambios a implementaciones concretas no siempre requieren cambios a las interfaces que implementan

        - Los buenos diseñadores de *software* y arquitectos trabajan mucho en reducir la volatilidad de las interfaces, intentando encontrar maneras de añadir funcionalidades a las implementaciones sin tener que hacer cambios en las interfaces

        - La implicación de esto es que las arquitecturas de *software* estables son aquellas que evitan depender de concreciones volátiles y que favorecen el uso de interfaces abstractas estables

    - La implicación de esto se puede expresar como un conjunto específico de prácticas de programación como las siguientes:

        - No se debe referir a clases concretas volátiles, sino a interfaces abstractas. Esto pone restricciones severas en la creación de objetos y generalmente obliga al uso de fábricas abstractas o *abstract factories*

        - No se tiene que derivar de clases concretas volátiles (relacionado con la herencia), sino que se tiene que derivar de interfaces abstractas. En lenguajes estáticos, la herencia es la relación más fuerte y rígida de todas las relaciones de código fuente, de modo que se debe usar con cuidado, mientras que, en los lenguajes dinámicos, la herencia es un problema menor (aunque sigue siendo una dependencia y se tiene que seguir teniendo cuidado)

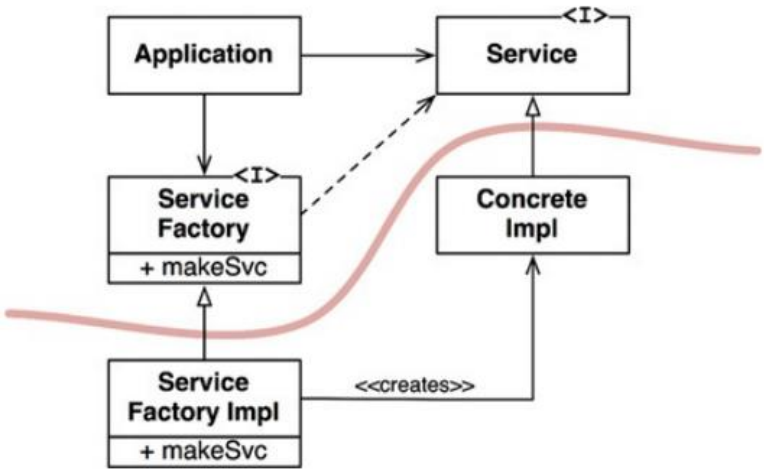
        - No se deben reescribir funciones concretas. Las funciones concretas normalmente requieren dependencias de código fuente, y cuando uno reescribe estas funciones, no se eliminan esas dependencias (porque se heredan), por lo que, para

gestionar estas dependencias se debe hacer la función abstracta y crear múltiples implementaciones

- Nunca se tiene que mencionar el nombre de algo concreto o volátil

- Para poder cumplir las reglas anteriores, la creación de objetos concretos volátiles requiere una gestión especial. Este cuidado se garantiza porque, en casi todos los lenguajes, la creación de un objeto requiere una dependencia de código fuente a la definición concreta del objeto

    - En la mayoría de lenguajes OO, tales como Java, uno usaría una *abstract factory* para gestionar la dependencia indeseable. La estructura de una *abstract factory* se enseña en el siguiente esquema:



    - En el ejemplo del esquema, $Application$ usa $ConcreteImpl$ a través de la interfaz $Service$. No obstante, $Application$ debe crear de alguna manera instancias de $ConcreteImpl$ sin crear dependencias de código fuente, por lo que llama al método $makeSvc$ de la interfaz $ServiceFactory$ (que se implementa en $ServiceFactoryImpl$, derivada de la clase $ServiceFactory$, que instancia $ConcreteImpl$ y lo devuelve como $Service$)

    - En este caso, se tiene que ver cómo se invierte el flujo de dependencia de código, dado que, de otro modo, $Application$ dependería de $ServiceFactoryImpl$ y se crearía $ConcreteImpl$ para $Service$, haciendo que hubiera una dependencia circular. No obstante, el flujo de control (de cómo se ejecuta) se queda intacto

    - La línea roja es una frontera arquitectónica que separa lo abstracto de lo concreto, y todas las dependencias de código fuente cruzan la curva en la misma dirección: hacia el lado

abstracto. La línea curva divide el sistema en un componente abstracto, el cual contiene todas las reglas de negocio de alto nivel, y el componente concreto, que contiene todos los detalles de implementación de estas reglas que se manipulan

- El componente concreto tiene una sola dependencia de código fuente, de modo que viola la DIP, pero esto es normal. Las violaciones de DIP no se pueden eliminar completamente, pero se pueden reducir a un pequeño número de componentes concretos y mantenerse separados del resto del sistema

  - La mayoría de sistemas contienen al menos un componente concreto de este tipo, normalmente llamado $main$ porque contiene la función principal

  - En el ejemplo, la función $main$ instanciaría $ServiceFactoryImpl$ y guardaría la instancia en una variable global del tipo $ServiceFactory$. $Application$ accedería después a la fábrica a través de la variable global

Los principios del diseño: cohesión de componentes

- Si los principios SOLID permiten saber cómo organizar los objetos dentro de un componente, los principios de componentes indican como organizar los componentes para crear un sistema de *software*

  - Los componentes son las unidades de despliegue, las entidades más pequeñas que se pueden desplegar como parte del sistema

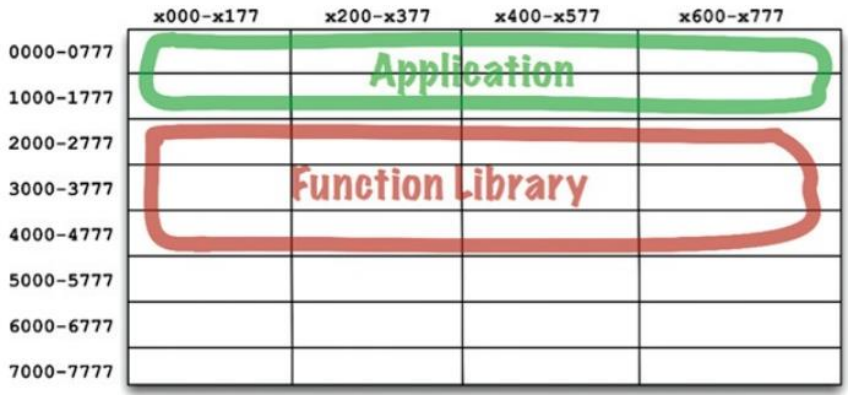
    - En Java, estos archivos son ".jar"; en Ruby, estos son archivos ".gem"; en lenguajes compilados, son agregaciones de archivos binarios; y en lenguajes interpretados, son agregaciones de archivos fuente

    - Los componentes se pueden vincular en un solo ejecutable, pueden estar agregados en un solo archivo o pueden desplegarse independientemente como *plugins* separados dinámicamente. Independientemente de como se despliegan, unos componentes bien diseñados tienen que retener la habilidad de ser desplegables independientemente, y por tanto, pueden desarrollarse independientemente

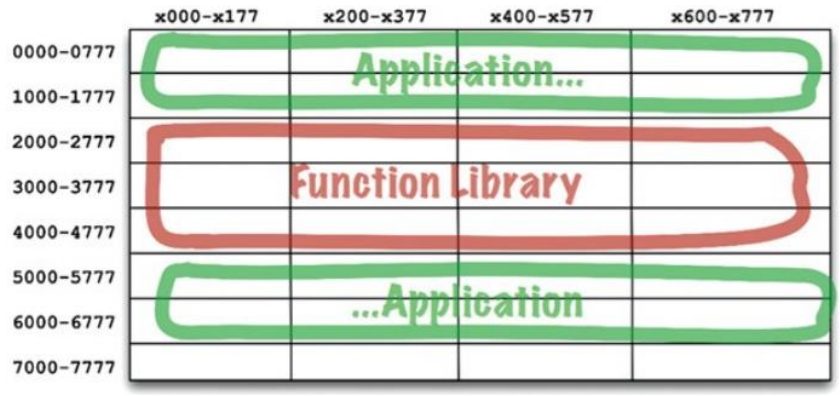
  - Hace mucho tiempo, los programadores debían incluir el código fuente de la librería de funciones con el código de la aplicación con y compilarlos en un solo programa con tal de acceder a la librería. Las librerías se mantenían en el código fuente, no en binario

- El problema con esto era que los dispositivos eran lentos y la memoria era limitada y cara, por lo que los compiladores tenían que hacer varios pasos sobre el código fuente, pero como la memoria era demasiado limitada para mantener todo el código residente, este tenía que leer el código fuente varias veces usando dispositivos lentos. Esto tomaba mucho tiempo, y cuanto más grande fuera la librería de funciones, más tiempo tardaba en compilarse

- Con tal de acortar los tiempos de compilación, los programadores separaron el código fuente de la librería de funciones del de las aplicaciones, compilando la librería separadamente y cargando el binario en una localización de memoria conocida. Cuando se quería ejecutar la aplicación, se cargaría la librería binaria y después se cargaría la aplicación, haciendo que la memoria se organizara como en el esquema:



- Esto funcionaba bien mientras que la aplicación se mantuviera dentro de unas localizaciones de memoria exactas, pero cuanto más grandes se hacían, los programadores tenían que dividirlas en dos segmentos de localizaciones distintas. Esto hacía que se tuviera que saltar alrededor de la librería de funciones, lo cual no era óptimo



- La solución a esta problemática fueron los archivos binarios relocalizables, que se basan en la idea de que el compilador se podía cambiar para resultar en códigos binarios que se pudieran relocalizar en la memoria por un cargador inteligente

    - Este cargador inteligente recibiría órdenes de donde cargar el código relocalizable. El código se instrumentaba con banderas o *flags* que indicaban qué partes de los datos cargados tenían que alterarse para ser cargados en una dirección seleccionada (esto solía significar que se tenía que añadir la dirección inicial a cualquier dirección de referencia de memoria en el archivo binario)

    - De este modo, el programador podía decirle al cargador dónde cargar la librería de funciones y dónde cargar la aplicación. El cargador aceptaría diversos insumos binarios y los cargaría en la memoria uno tras otro, relocalizándolos y cargándolos (permitía cargar solo aquellas funciones que se necesitaran)

    - También se cambió el compilador para emitir los nombres de las funciones como metadatos en un binario relocalizable. Si un programa llamaba a la librería, el compilador emitiría el nombre como una referencia externa, mientras que si el programa definia una librería, se emitiría como definición externa

    - Después, el cargador podía vincular las referencias externas a las definiciones externas una vez que haya determinado dónde ha cargado estas definiciones

- El cargador vinculante o *linking loader* permitió a los programadores dividir sus programas en segmentos compilables y cargables por separado. Esto funcionaba cuando las librerías y los programas eran relativamente pequeños, pero estos comenzaron a crecer mucho con el tiempo

    - Eventualmente, los cargadores vinculantes eran demasiado lentos, por lo que se comenzó a separar la carga y la vinculación en dos fases. Los programadores tomaron la parte lenta que se encargaba de la vinculación en una aplicación separada llamada vinculador o *linker*, de modo que esta aplicación resultaba en un relocalizable vinculado que un cargador podía cargar muy rápidamente

    - A partir de los finales de los 80, los discos se volvieron más rápidos y la memoria de los ordenadores comenzó a crecer, de modo que la vinculación tardaba cada vez menos y menos y se pudo volver a hacer la vinculación durante el tiempo de carga

- Una de las decisiones más importantes es saber qué clases pertenecen a qué componentes, y se necesitan principios buenos para guiar la ingeniería de *software*

    - El principio REP (de *Reuse/Release Equivalence Principle*) es un principio que se basa en que aquellos que deseen reutilizar componentes de software no podrán hacerlo, y no lo harán, a menos que esos componentes sean rastreados a través de un proceso de lanzamiento y se les asignen números de lanzamiento

        - Esto no es simplemente porque, sin números de lanzamiento, no habría otra manera de asegurar que todos los componentes reusados son compatibles el uno con el otro. Más bien, este refleja el hecho de que los desarrolladores necesitan saber cuando los lanzamientos van a llegar, y qué cambios traerán consigo estos nuevos lanzamientos

        - No es raro que algunos desarrolladores sepan de la llegada de un nuevo lanzamiento y, dado los cambios que este introduce, decidan continuar usando un lanzamiento antiguo. Por lo tanto, el proceso de lanzamiento debe producir notificaciones apropiadas y documentación para que los usuarios puedan tomar decisiones informadas sobre si deben o cuando integrar el nuevo lanzamiento

        - Desde el punto de vista del diseño y la arquitectura, este principio significa que las clases y módulos que formen parte de un componente deben pertenecer a un grupo cohesivo. Un componente no puede simplemente consistir de una mezcla de clases y módulos, sino que debe haber una temática o propósito común que todos los módulos comparten

        - Aunque esto parezca obvio, esto implica que las clases y módulos que se agrupan juntos deberían ser capaces de lanzarse juntos. El hecho de que compartan el mismo número de versión y el mismo seguimiento (y que están incluidos bajo la misma documentación) debería tener sentido para todos

    - Igual que el SRP dice que una clase no debería contener múltiples razones para cambiar, el principio CCP (de *Common Closure Principle*) expresa que un componente no debería tener diferentes razones para cambiar

        - Para la mayoría de aplicaciones, la manutención es más importante que la reusabilidad: si el código en una aplicación debe cambiar, uno querría que todos esos cambios ocurrieran solo en un componente y que no estén distribuidos en varios

componentes. Si los cambios se confinan a un solo componente, solo se necesita desplegar otra vez ese componente, y los otros componentes que no dependen del componente cambiado pueden seguir igual

- El CCP dice que se tiene que reunir en un solo lugar todas las clases que puedan cambiar por las mismas razones: si dos clases están tan estrechamente relacionadas que siempre cambian con conjuntamente, entonces deben pertenecer al mismo componente. Esto minimiza la carga de trabajo relacionada al lanzamiento, la validación y el despliegue del *software*

- Este principio está estrechamente asociado con el OCP, dado que se dice que las clases deberían estar cerradas para la modificación, pero abiertas para la extensión. El CCP amplifica esto haciendo que se junten las clases que estén cerradas bajo el mismo tipo de cambios bajo el mismo componente, haciendo que los cambios se restrinjan a un mínimo número de componentes

- El CCP, al igual que el SRP dice que se tienen que separar métodos en diferentes clases si cambian por diferentes motivos, dice que se tienen que separar las clases en diferentes componentes si se cambian por diferentes razones

- El CRP (de *Common Reuse Principle*) es otro principio que ayuda a decidir qué clases y módulos deberían ponerse dentro de un componente. Este expresa que las clases y módulos que tienden a reutilizarse juntos deberían pertenecer al mismo componente

    - Las clases no se suelen reutilizar en solitario, dado que colaboran con otras clases que deben ser parte de la misma abstracción reutilizable. El CRP dice que estas clases deberían pertenecer al mismo componente, de modo que se debería esperar que dentro de este componente haya clases con muchas dependencias entre ellas

    - Un simple ejemplo puede ser una clase contenedora que se asocia con sus iteradores. Estas clases se tienen que reutilizar a la vez porque se usan juntas, por lo que deberían estar en el mismo componente

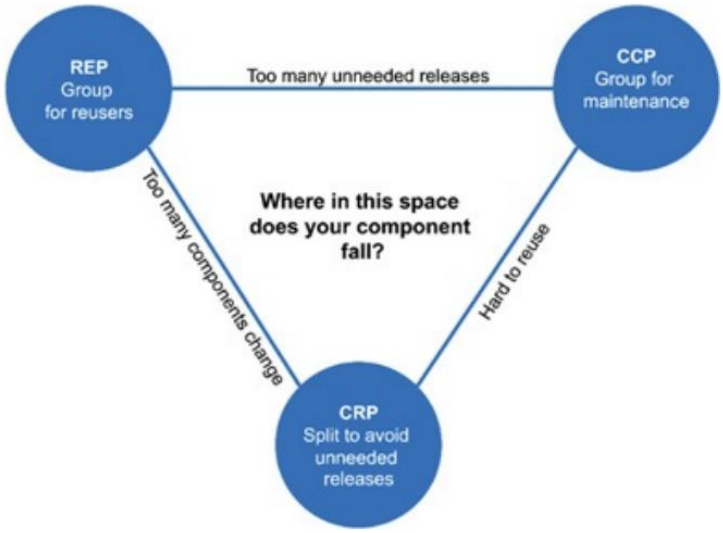
    - No obstante, el CRP también indica qué clases no juntar en un componente: cuando un componente usa otro, se crea una dependencia entre componentes, aunque solo utilice una clase del componente del que se depende. Por lo tanto, cuando un componente depende de otro, se querría que dependa de todas

las clases en el otro componente, de modo que no se necesita redesplegar más veces de lo necesario

- El CRP es una versión genérica del ISP: el ISP aconseja que no haya dependencias a clases con métodos que no se usan, y el CRP dice que no haya dependencias de componentes de clases que no se usan

- Sin embargo, estos tres principios tienden a estar en conflicto entre ellos: el REP y el CCP son principios inclusivos (hacen que los componentes se agranden), mientras que el CRP es exclusivo (hace que los componentes se reduzcan). Es la tensión entre estos tres principios lo que un bun arquitecto busca resolver (quiere encontrar un equilibrio)

  - El siguiente diagrama es un diagrama de tensión que muestra como los tres principios de cohesión interactúan el uno con el otro. Los arcos del diagrama describen el coste de abandonar un principio en el vértice opuesto



  - Un arquitecto que solo se centra en el REP y CRP encontrará que habrá muchos componentes impactados cuando se realizan cambios simples, mientras que uno que solo se centre en el CCP y el REP causaría que se generen demasiados lanzamientos innecesarios

  - Un buen arquitecto intenta encontrar una posición en ese triángulo de tensión que permita resolver las preocupaciones actuales del equipo de desarrollo, pero sabiendo que estas preocupaciones pueden cambiar con el tiempo

  - En general, los proyectos tienden a comenzar en el lado derecho del triángulo (donde la única preocupación es la reusabilidad), pero cuando el proyecto madura, el proyecto tiende a ir a la

derecha, por lo que la estructura del componente varía con el tiempo y la madurez. Esto tiene que ver más con cómo se desarrolla y se usa el proyecto que lo que realmente hace el proyecto

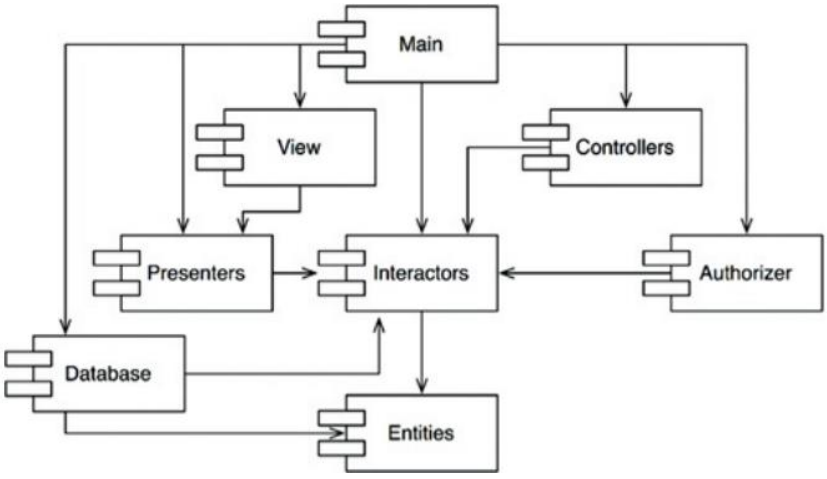
Los principios del diseño: relación entre componentes

- Los principios anteriores lidiaban con la cohesión dentro de los componentes, pero también se tienen principios para poder establecer la relación entre los diferentes componentes. El primer principio que se revisa es el principio ADP (de *Acyclic Dependencies Principle*), que se basa en la idea de que no debe haber ciclos de dependencia entre componentes
    - Un problema usual en entornos de desarrollo es aquel en el que varios desarrolladores están modificando el mismo código fuente y algunos cambios se sobreponen, haciendo que haya piezas del sistema que ya no funcionen debido a esos cambios
        - Aunque esto no es un problema grave para equipos de desarrollo pequeños, cuando el proyecto y el equipo crecen, esto se vuelve un problema más probable
        - Durante las últimas décadas se han desarrollado dos soluciones: el *weekly build* y el principio ADP
    - El *weekly build* solía ser común en proyectos medianos, y funciona de la siguiente manera: todos los desarrolladores se ignoran el uno al otro en los primeros cuatro días de la semana (trabajan en copias privadas del código, sin integrar cambios) y en el último día, todos integran sus cambios para construir el sistema
        - Aunque esto tiene la ventaja de que los desarrolladores pueden trabajar independientemente cuatro días a la semana, la desventaja más grande es el precio que se paga el último día
        - Cuanto más grande se hace el proyecto, esto se hace menos asequible, dado que la integración tomará más tiempo. Como se necesitará acortar el tiempo de trabajo autónomo, la eficiencia disminuirá y se llegará a una crisis
    - La solución a este problema es partir el entorno de desarrollo en componentes lanzables, de modo que los componentes se vuelven unidades de trabajo que pueden ser responsabilidad de un solo desarrollador o equipo. Cuando uno de los componentes ya funciona, se lanza para que los otros desarrolladores puedan usarlo, y se siguen modificando en privado para seguir produciendo lanzamientos y nuevas versiones

- Cuando un nuevo lanzamiento está disponible, los otros equipos o desarrolladores pueden decidir si usarlo al momento o esperar a desarrollar otras cosas con la versión antigua y después actualizarse

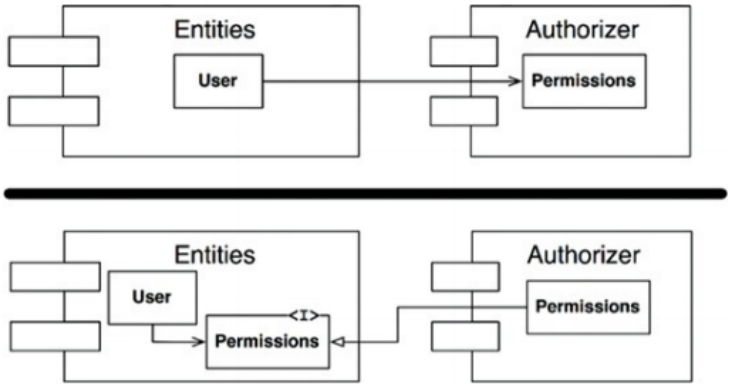
- De este modo, ningún equipo está a merced de otro: los cambios hechos a cada componente no necesariamente tienen un efecto directo en sus propios componentes. Además, no hay un único punto en el tiempo en el que los desarrolladores deban integrar todos los cambios que han ido haciendo

- Para hacer que este proceso funcione uno se debe asegurar de que no hay ciclos de dependencia, de modo que el grafo que muestra la estructura de dependencia (con flechas de dependencia de los componentes) es dirigido y acíclico. De este modo, aquellos componentes que si dependan de uno concreto podrán decidir si integrar los cambios al instante o no, pero si no hay ciclos, habrá componentes que no se vean afectados en casi nada

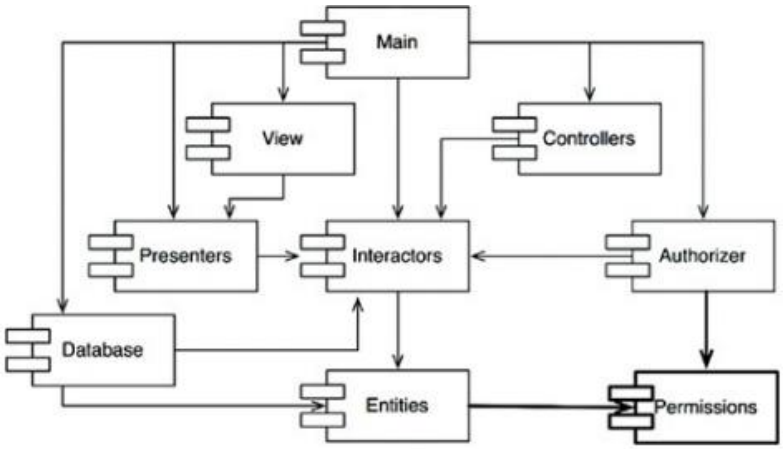


- La existencia de un ciclo haría que hubiera un alto nivel de dependencia entre diferentes componentes y posiblemente haría que los cambios tuvieran un efecto en otros componentes menos relacionados. Además, cuando hay ciclos es más difícil saber el orden en el que se tienen que construir los diferentes componentes

- Una manera para romper un ciclo de dependencia es a través del DIP, de modo que se puede crear una interfaz que permita invertir la dependencia entre clases y así conseguir que la dependencia entre componentes vaya en la dirección deseada



- Otra manera de romper el ciclo es a través de crear un nuevo componente del cual dependan dos componentes dentro del ciclo, de modo que se amplía la estructura, pero se rompe el ciclo. Esta solución, no obstante, hace que la estructura sea volátil



- Los problemas anteriores muestran que no se puede diseñar la estructura de los componentes de arriba abajo o *top-down*: no es una de las primeras cosas cuando se diseña el sistema, sino que evoluciona cuando el sistema crece y cambia

  - Cuando se ve una agrupación de una alta granularidad como la estructura de dependencia de los componentes, uno cree que los componentes deberían representar de alguna manera las funciones del sistema. No obstante, este no es un atributo de los diagramas de dependencia de los componentes (no describen la función de la aplicación)

  - Los diagramas son un mapa para la constructibilidad y manutención de la aplicación, por lo que no se diseñan al principio del proyecto (no hay ningún *software* a construir o mantener aún)

  - Cuando se van añadiendo módulos y el proyecto crece en tamaño, entonces ya hay una necesidad de poder hacer seguimiento de las dependencias

        - Una de las preocupaciones principales con la estructura de dependencia es el aislamiento de la volatilidad, dado que no se quiere que los componentes cambien frecuentemente. Por lo tanto, el diagrama se modela para proteger la estabilidad de los componentes de alto valor frente a aquellos más volátiles

        - Cuando la aplicación crece, entonces uno comienza a preocuparse por la reusabilidad, de modo que el CRP comienza a influir en la composición de componentes. Finalmente, cuando los ciclos aparecen, el ADP se aplica y la dependencia de componentes crece

- El segundo principio es el principio SDP (de *Stable Dependencies Principle*), que se basa en la idea de que las dependencias entre componentes deben ser estables

    - Los diseños no pueden ser completamente estáticos, sino que hay un nivel de volatilidad necesaria para que se mantenga el diseño. Respetando el CCP, se crean componentes que son sensibles a unos tipos de cambio, pero inmunes a otro, por lo que los componentes se diseñan para que haya una cierta volatilidad

        - Cualquier componente que se espere que sea volátil no debería depender de un componente que sea difícil de cambiar, o el componente volátil también sería difícil de cambiar

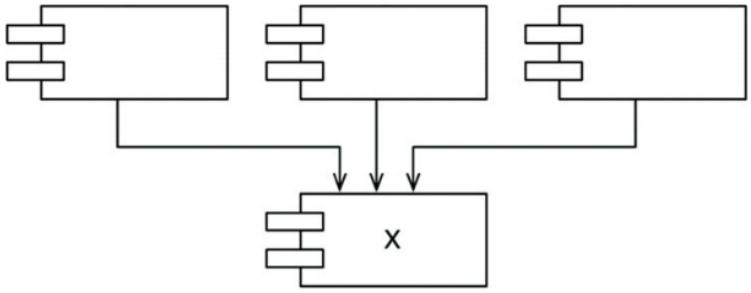
        - Cuando uno sigue el principio SDP (de *Stable Dependencies Principle*), uno se asegura que los módulos que se supone que son fáciles de cambiar no dependen de módulos difíciles de cambiar

    - La estabilidad a la que uno se refiere en este principio tiene relación con la cantidad de trabajo que requiere realizar un cambio: tirar una moneda en equilibrio es más difícil que tirar una mesa

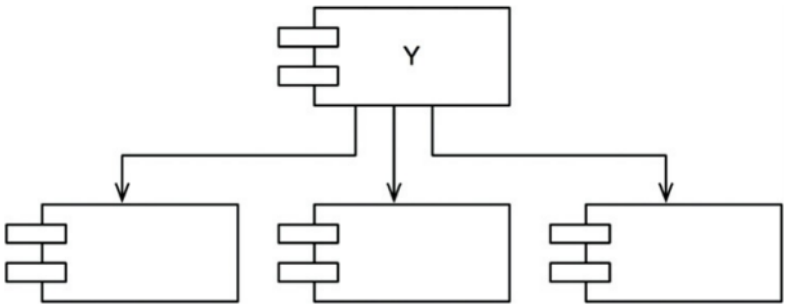
        - Hay muchos factores que hacen que un componente de *software* sea difícil de cambiar, como su tamaño, su complejidad, su claridad, y otros. Ignorando estos factores, una de las maneras más seguras de hacer un componente difícil de cambiar es que dependa de varios otros componentes: un componente con muchas dependencias es muy estable porque requiere mucho trabajo reconciliar cualquier cambio con todos sus componentes dependientes

        - El diagrama siguiente muestra que cuando varios componentes dependen de uno solo, este es responsable de estos

componentes y que es muy estable por ello. Si este componente no depende de ningún otro, se dice que este componente es independiente



- El diagrama siguiente, en cambio, muestra que un componente no es estable debido a que depende de varios componentes a la vez, haciendo que el componente sea irresponsable si no depende de otro y dependiente



- Finalmente, el tercer principio de relación de componentes que se estudia es el principio SAP (de *Stable Abstractions Principle*), que se basa en la idea de que hay una relación entre el nivel de abstracción de un componente y su estabilidad

  - Hay partes del *software* del sistema que no deben cambiar muy seguido, el cual representa una arquitectura de alto nivel y decisiones sobre la política (no se quiere volatilidad)

    - Por lo tanto, aquel *software* que encapsule estos elementos debería ponerse en componentes estables. Los componentes inestables solo deberían contener *software* volátil (que se puede cambiar rápido y fácil)

    - No obstante, si las políticas de alto nivel se ponen en componentes estables, entonces serán difíciles de cambiar, haciendo que la arquitectura entera sea inflexible. Para poder lidiar con esto se acude al OCP, de modo que se pueden extender clases abstractas sin necesidad de modificarse

  - El SAP establece una relación entre la estabilidad y la abstracción: por un lado, dice que un componente estable debería ser abstracto con tal de

que su estabilidad no evite que se pueda extender, mientras que, por otro, dice que un componente inestable debe ser concreto debido a que su inestabilidad hace que el código concreto sea fácil de cambiar

- Por lo tanto, si un componente es estable, debería consistir de interfaces y clases abstractas que sean extensibles. Los componentes estables que son extensibles son flexibles y no restringen la arquitectura

- El SAP y el SDP combinados son equivalentes al DIP pero para componentes, de modo que las dependencias se mueven en la dirección de las abstracción. No obstante, el DIP lidia con clases, que por naturaleza son o no son abstractas, pero cuando se habla de componentes, estos pueden ser más o menos abstractos o estables

La arquitectura: independencia, fronteras, reglas y políticas

- La arquitectura de un sistema de *software* es la forma dada a ese sistema por aquellos que lo han construido. La forma es la división de ese sistema en componentes, la organización de estos componentes y la manera en la que estos se comunican entre ellos

    - El propósito de la forma es facilitar el desarrollo, el despliegue, la operación, y la manutención del sistema de *software* contenido en él. La estrategia detrás de esta facilitación es dejar la mayor cantidad de opciones abiertas como sea posible, el tiempo que sea posible

        - Aunque se quiere que el sistema funcione correctamente y adecuadamente, la arquitectura no tiene que ver tanto con la operativa del sistema, sino que tiene más que ver con su despliegue, manutención y desarrollo posterior. No obstante, esto no quiere decir que la arquitectura no tenga que ver con el funcionamiento del sistema, solo que este rol no es crítico

        - El propósito principal de la arquitectura es apoyar el ciclo de vida del sistema: una buena arquitectura hace que el sistema sea fácil de entender, desarrollar, mantener y desplegar. El objetivo final es minimizar el coste de la vida útil del sistema y maximizar la productividad de los programadores

    - Un sistema de *software* que es difícil de desarrollar no es probable que tenga una vida larga y sana, de modo que la arquitectura de un sistema debe hacer que el desarrollo sea fácil

- Las diferentes estructuras del equipo implican diferentes decisiones arquitectónicas, dado que se tienen que adaptar a cómo desarrollarán el sistema con los recursos disponibles

- Esto es más obvio cuando se considera cómo equipos pequeños de programadores no suelen definir bien componentes o interfaces para no impedir el progreso los primeros días. En cambio, equipos grandes necesitan que el sistema esté bien dividido en componentes y que tenga interfaces estables para poder desarrollar el sistema

- La arquitectura de un equipo por componente no suele ser la mejor arquitectura para el despliegue, la operación y la manutención, pero es una estructura común cuando solo se tiene en cuenta el esquema de desarrollo

- Para que un sistema de *software* sea efectivo, debe de ser desplegable: cuanto mayor es el coste de despliegue, menor es la utilidad del sistema. El objetivo de una buena arquitectura de *software* es hacer que el sistema se pueda desplegar fácilmente con una sola acción

    - Desafortunadamente, la estrategia de despliegue no se suele considerar durante el desarrollo inicial, lo cual lleva a arquitecturas que hacen que el sistema sea fácil de desarrollar, pero difícil de desplegar

    - Por ejemplo, en el desarrollo inicial del sistema, los desarrolladores pueden decidir el uso de una arquitectura de microservicios, lo cual puede tener sentido porque hace que el sistema sea fácil de desarrollar por las fronteras firmes entre componentes y las interfaces relativamente estables. No obstante, cuando se despliega el sistema, el número de microservicios es grande y configurar las conexiones entre ellos y su inicialización pueden comportar varios errores

- El impacto de la arquitectura en la operación del sistema tiende a ser menos dramático que su impacto en los otros aspectos. Casi cualquier dificultad operacional se puede resolver usando más *hardware* en el sistema sin drásticamente impactar en la arquitectura de *software*

    - Los sistemas de *software* que tienen arquitecturas ineficientes pueden ser más efectivos si se añade más memoria y más servidores. El hecho de que el *hardware* sea menos costoso y la mano de obra sea más cara hace que los problemas de arquitectura con la operativa no sean tan problemáticos como aquellos relacionados con los otros aspectos

- Una buena arquitectura de *software* comunica las necesidades operacionales del sistema, de modo que la operación del sistema es aparente para los desarrolladores. La arquitectura debería elevar los casos de uso, las características y los comportamientos requeridos del sistema a los desarrolladores, de modo que el sistema sea más fácil de entender

- De todos los aspectos del sistema de *software*, la manutención es la parte más costosa: las infinitas características nuevas que se pueden añadir y los defectos inevitables y correcciones posteriores consumen muchos recursos

    - El coste principal de la manutención proviene de la espeleología y el riesgo: la espeleología es el coste de adentrarse en el *software* para tratar de determinar el mejor lugar y la mejor estrategia para añadir una nueva característica o reparar un defecto. Cuando se hace esto, la probabilidad de crear defectos inadvertidos hace que haya un coste por este riesgo

    - Una buena arquitectura mitiga estos costes: separando el sistema en componentes y aislando estos a través de interfaces estables, es posible crear maneras para añadir características futuras y reducir el riesgo de errores nuevos

- El *software* tiene dos tipos de valor: el valor de su comportamiento y el valor de su estructura, y es el segundo valor el que hace que el *software* sea suave. Este se creo para poder cambiar rápida y fácilmente el comportamiento de las máquinas, pero la flexibilidad depende de la forma del sistema, la organización de los componentes y las interconexiones entre ellos

    - La manera de hacer que el *software* siga siendo suave es a través de mantener las opciones lo más abiertas posibles, el mayor tiempo posible. Las opciones que se quieren dejar abiertas son todos aquellos detalles que no importa

    - Todos los sistemas de *software* pueden estar descompuestos en dos elementos principales: la política y los detalles. El elemento de la política tiene las reglas de negocio y los procedimientos (donde reside el verdadero valor del sistema), mientras que los detalles son aquellas cosas necesarias para que individuos u otros sistemas puedan comunicar la política, pero sin afectar al comportamiento de esta (dispositivos IO, bases de datos, etc.)

    - El objetivo del arquitecto es crear una forma para el sistema que reconozca la política como el elemento más esencial del sistema

mientras que los detalles se mantienen irrelevantes a aquella política. Esto permite que las decisiones sobre los detalles se atrasen para otro momento: si uno puede desarrollar una política de alto nivel sin comprometerse a los detalles que la rodean, se pueden atrasar las decisiones para poder hacerlas con más información

        - Esto, a su vez, permite hacer más experimentos para probar cosas diferentes y recolectar información para una mejor toma de decisiones. Un arquitecto bueno maximiza el número de decisiones no tomadas aún

    - Uno de los errores más comunes hace años era realizar programas en un lenguaje específico que dependía del dispositivo, lo cual hacía muy difícil mover el programa de un dispositivo a otro

        - A partir de esta problemática nación la independencia del dispositivo: los sistemas operativos abstrajeron los dispositivos IO en funciones de *software*, de modo que los operadores podían comunicar al sistema operativo si un programa debía conectarse a uno u otro dispositivo concreto sin tener que cambiar el programa

- Existen varios aspectos y preocupaciones que una buena arquitectura debería tener en cuenta y equilibrar a través de su diseño, de modo que las partes se puedan tratar de manera independiente

    - Que un sistema apoye los casos de uso significa que la arquitectura del sistema debe apoyar el propósito del sistema. Esta es la primera preocupación del arquitecto y su principal prioridad

        - No obstante, la arquitectura no tiene un efecto muy importante sobre el comportamiento del sistema: hay pocas opciones del comportamiento que la arquitectura puede dejar abierta. Pero la influencia no lo es todo, y lo más importante que una buena arquitectura puede hacer para apoyar el comportamiento es clarificar y exponer el comportamiento

        - De este modo, el propósito principal del sistema será visible a nivel arquitectónico. Los desarrolladores no tendrán que buscar comportamientos, dado que estos serán fácilmente visibles, y los elementos (clases y funciones) se podrán describir fácilmente

    - La arquitectura juega un rol más sustancial en el apoyo a la operativa del sistema: la arquitectura debe soportar el tipo de *throughput* y tiempo de respuesta que demanda cada caso de uso

        - Esta decisión es una de las opciones que un buen arquitecto deja abierta: un sistema que se escribe para que sea fijo y con estructura fija no puede ser actualizado a múltiples procesos, *threads* o microserviocios cuando nace la necesidad

        - En comparación, una arquitectura que mantiene un aislamiento adecuado de sus componentes y que no asume los medios de comunicación entre estos componentes será más fácil de transicionar al espectro de los *threads*, procesos y servicios que puede llegar a requerir la operativa del sistema en el futuro

    - La arquitectura también tiene un papel significativo en el apoyo del entorno de desarrollo. Cualquier organización que diseñe un sistema producirá un diseño cuya estructura es una copia de la estructura de comunicación de la organización (ley de Conway)

        - Un sistema que se debe desarrollar por una organización con muchos equipos y preocupaciones tendrá una estructura que facilite las acciones independientes y que no interfieran durante el desarrollo

        - Esto se cumple a través de partir adecuadamente el sistema en componentes bien aislados e independientemente desplegables. Los componentes, entonces, pueden ser asignados a equipos que pueden trabajar en ellos independientemente

    - La arquitectura también tiene un papel importante en determinar la facilidad con la que el sistema se desplegará. El objetivo es el despliegue inmediato

        - Una buena arquitectura no se apoya en docenas de *scripts* de configuración y modificaciones de archivos propietarios, o un manual de instrucciones para la creación de diversos archivos, sino que se puede desplegar justo después de ser construido

        - Esto se consigue a través de partir y aislar los componentes del sistema, incluyendo aquellos componentes maestros que ligan el sistema entero junto y aseguran que cada componente se inicia, se integra, y se supervisa adecuadamente

    - Aunque una buena arquitectura balancea todas las preocupaciones anteriores con una estructura de componentes que satisfaga todo, este equilibrio es muy difícil de conseguir

        - El problema es que no se suelen saber todos los casos de uso, las restricciones operativas, la estructura del equipo o los

requerimientos de despliegue. Estos aspectos también cambian a lo largo del ciclo de vida del sistema

- Aunque existan problemas para equilibrar todas las preocupaciones, hay principios de arquitectura no muy costosos de implementar y que pueden ayudar a equilibrar todas estas preocupaciones y problemas. Estos permiten partir el sistema en componentes bien aislados y dejan opciones abiertas

    - Aunque el arquitecto no sepa todos los casos de uso del sistema, sí que sabe el propósito principal del sistema, por lo que puede aplicar el SRP y el CCP para separar cosas que cambian por diferentes motivos y agrupar aquellas que cambian por los mismos motivos

        - Las cosas que cambian por diferentes razones pueden ser cosas obvias como la interfaz del usuario, ya que las reglas de negocio no cambian, pero los casos de uso tienen cosas de ambos elementos. Por lo tanto, se tienen que separar aquellas porciones que cambian por un tipo de razones de las otras porciones, de modo que se puedan cambiar independientemente

        - Las reglas de negocio mismas pueden estar muy relacionadas a la aplicación o ser más generales. Estos dos tipos de reglas pueden cambiar por diferentes razones, de modo que deberían estar separadas para cambiar independientemente

        - La base de datos, el lenguaje de *querys* y el esquema de detalles técnicos, por ejemplo, no tendrían nada que ver con cosas como las reglas de negocio o la interfaz de usuario, de modo que se debería separar esta parte del resto del sistema para que se cambie independientemente

        - Por lo tanto, se puede ver que el sistema se puede dividir en capas horizontales disasociadas o *decoupled layers*

    - Los casos de uso también cambian por diferentes razones, por lo que estos tienen que estar divididos

        - Estos son particiones verticales que cortan a través de las capas horizontales del sistema, dado que cada caso de uso tendrá elementos asociados de las reglas de negocio, funcionalidades, interfaces, etc. Por lo tanto, se divide el sistema en capas horizontales, pero también en casos de uso horizontales que cortan a través de estas capas

        - Para conseguir que se desasocien las capas, se separan los diversos elementos de cada una de las capas horizontales, de modo que se divide el sistema en varias agrupaciones

dependiendo del caso de uso y las razones de cambio. De este modo, se pueden afectar las diversas partes del sistema de manera independiente

- Si los diferentes aspectos de los casos de uso están separados, entonces aquellos que deben ejecutarse a un *throughput* alto deben de estar probablemente separados de aquellos que deben ejecutarse a un *throughput* bajo (pueden ejecutarse en servidores diferentes)

    - La disociación que se ha hecho para los casos de uso ha ayudado con la disociación en relación a las operaciones. Sin embargo, para poder aprovechar el beneficio operacional, esta disociación debe tener el modo correcto

    - Cuando se ejecuta en servidores separados, los componentes separados no pueden depender de estar juntos en el mismo espacio de direcciones de un procesador (intervalo de direcciones discretas de almacenamiento de datos y programas). Estos deben ser servicios independientes, que comunican sobre una red de algún tipo

    - Muchos arquitectos llaman a estos componentes "servicios" o "microservicios", dependiendo de una noción vaga del número de líneas. El punto importante es que hay veces que se tienen que separar los componentes hasta el nivel de servicio

- La disociación de los componentes también tiene un impacto en la habilidad de desarrollo y en la de despliegue

    - Claramente, cuando los componentes están fuertemente disociados, la interferencia entre equipos se mitiga. De este modo, mientras que las capas y los casos de uso estén disociados, la arquitectura del sistema apoyará la organización de equipos, independientemente de si están organizados de una manera u otra

    - La disociación de los casos de uso y capas también permite que sea asequible un alto nivel de flexibilidad en el despliegue, de modo que es posible intercambiar capas y casos de uso en los sistemas de ejecución

- Los arquitectos a veces caen en el error de la duplicación de código, habiendo diferentes tipos de duplicación

    - La duplicación verdadera es aquella en la que cada cambio de una instancia necesita el mismo cambio en cualquier duplicado de la instancia. La duplicación falsa o accidental ocurre cuando dos

piezas de código aparentemente duplicadas evolucionan en diferentes caminos (de modo que no son duplicados verdaderos)

- Cuando uno separa verticalmente los casos de uso o horizontalmente las capas, uno se encontrará con la tentación de duplicar un código para dos casos de uso muy similares. No obstante, uno tiene que ser muy precavido y no unificar código sin antes asegurarse que es una duplicación verdadera (de modo que se puede eliminar el otro código y usar solo uno de los dos), dado que después puede ser difícil separarlos

- Hay varios modos de desasociar los casos de uso y las capas: estas pueden estar desasociadas a nivel de código fuente, a nivel de código binario y a nivel de unidad de ejecución (servicio)

  - A nivel de código fuente, uno puede controlar las dependencias entre los códigos de fuente de módulos, de modo que los cambios en un módulo no fuercen la recompilación de los otros. En este modo, los componentes se ejecutan en el mismo espacio de direcciones y se comunican llamando a funciones (solo hay un ejecutable cargado en la memoria del computador)

  - A nivel de despliegue, uno puede controlar las dependencias entre unidades desplegables a través de archivos especiales o librerías compartidas, de modo que los cambios en el código fuente de un módulo no fuercen a los otros módulos a reconstruirse y redesplegarse. Muchos de los componentes seguirán en el mismo espacio de direcciones y se comunicarán con llamadas, pero otros componentes pueden estar en otros procesos del mismo procesador y comunicarse entre comunicaciones interprocesales, *sockets* (estructura de *software* dentro de una red de computadora que sirve como punto externo para recibir y enviar datos a través de la red) o memoria compartida

  - A nivel de servicio, se puede reducir las dependencias hasta el nivel de las estructuras de datos, y comunicarse solamente a través de la red de modo que cada unidad de ejecución sea enteramente independiente del código fuente y el código binario de otros (como con servicios y microservicios)

- Saber cuál de estos modos de disociación es el mejor es difícil durante las primeras fases de un proyecto, y el óptimo puede cambiar. Una solución es simplemente disociar a nivel de servicio por defecto, aunque esto es caro (en recursos y tiempo) y tiende a que se haga una disociación muy granular

        - Una manera es empujar la disociación hasta el punto en el que el servicio puede formarse un servicio cuando sea necesario, pero dejar los componentes en el mismo espacio de direcciones el mayor tiempo posible. Esto hace que la opción del servicio siga abierta

        - Con este enfoque, los componentes iniciales se separan en el nivel del código fuente, pero que, si surgen problemas de despliegue o desarrollo, impulsar la disociación a nivel del despliegue puede ser suficiente. Si los problemas siguen aumentando, se puede escoger cuidadosamente qué unidades desplegables convertir en servicios y gradualmente pasar el sistema a este modo de disociación

        - Si los problemas decrecen en el tiempo y solo se necesita disociación a nivel del código fuente o del despliegue, entonces se puede progresar a ese modo de disociación

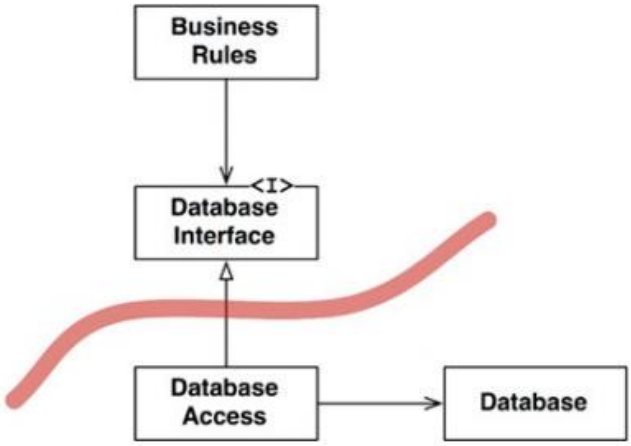
- La arquitectura de *software* es el arte de dibujar fronteras para separar elementos de *software* entre ellos y restringir aquellos de un lado a que sepan una información concreta de los otros elementos fuera de la frontera. Esta arquitectura se define por un conjunto de componentes y fronteras de diferentes tipos

    - Uno establece fronteras entre las cosas que importan y las que no importan para diferentes componentes

        - Por ejemplo, la GUI no importa para las reglas de negocio, por lo que habría una línea que separa ambas cosas, del mismo modo que debería haber una línea entre la GUI y la base de datos

        - Aunque algunas cosas pueden estar relacionadas y pareciera que no debería haber una frontera separadora, se tiene que ver que algunos elementos del sistema son usados por otros de manera indirecta (no se necesitan saber los detalles de estos elementos)

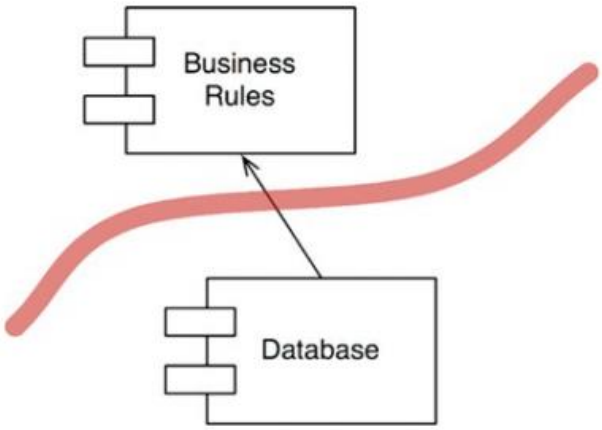
    - En el siguiente ejemplo, se puede ver como $BusinessRules$ depende de la interfaz $DatabaseInterface$ y no de la clase $DatabaseAccess$ que la implementa. Además, la interfaz y la base de datos no saben que existe la clase $DatabaseAccess$ (porque esta depende de esas clases), y la frontera se fija en medio de la relación de herencia



- Viendo el componente que contiene las reglas de negocio y el componente que contiene la base de datos y las clases de acceso, se puede ver como la relación de dependencia va del segundo componente al primer componente. Esta dirección indica que la base de datos no importa para las reglas de negocio, pero esta no puede existir sin las reglas de negocio

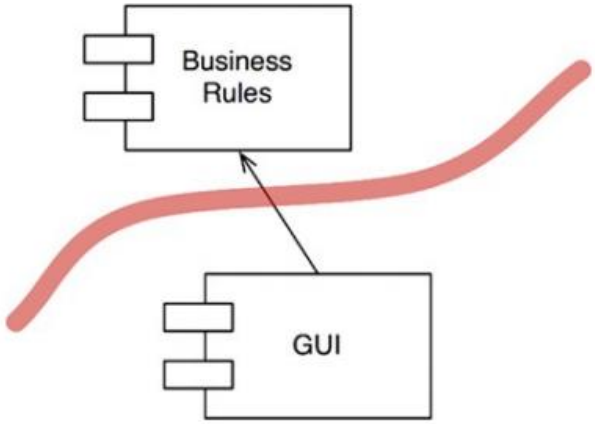
- De este modo, las reglas de negocio pueden usar cualquier base de datos (diferentes implementaciones) sin que tenga importancia para el componente de las reglas de negocio. Es la traducción de las llamadas a la base de datos es lo único de lo que depende el componente de las reglas de negocio, pero no de la implementación exacta

- Por lo tanto, la frontera entre componentes se establece entre la relación de dependencia de código fuente de la base de datos a las reglas de negocio



- Los desarrolladores y clientes a veces se confunden y se olvidan de uno de los principios más importantes: el IO es irrelevante. Normalmente se piensa en el comportamiento del sistema en términos del comportamiento del IO, pero uno se olvida del

modelo (reglas de negocio) subyacente y que este no necesita la interfaz, por lo que los componentes relacionados con los dispositivos IO deben estar separados



- Tomando en cuenta la decisión sobre la base de datos y la GUI, se crea un patrón para la adición de otros componentes para que se permita a un sistema añadir *plugins* de terceras partes

    - De este modo, el sistema se puede escalar debido a que las reglas de negocio se separan (y son independientes de) los componentes que son opcionales o que se pueden implementar de diferentes formas

    - La relación de dependencia vista entre componentes es asimétrica (el componente del que depende otro tiene poder para alterar la compatibilidad, pero el componente dependiente no), y esto es deseable en los sistemas. Uno quiere que haya componentes inmunes a otros, de modo que cambios en una parte no causen un efecto en partes no relacionadas del sistema

    - Organizar el sistema con una arquitectura de *plugin* crea *firewalls* entre los cambios para que no se propaguen, estableciendo fronteras. Estas fronteras se fijan en donde haya un eje de cambio y permite que se diferencien elementos que cambien a diferentes velocidades y por diferentes razones (el principio SRP)

- En el tiempo de ejecución, un cruce de fronteras no es más que una función de un lado de la frontera llamando a otra del otro lado de la frontera y pasando algunos datos. El truco para crear un cruce apropiado es a través de gestionar las dependencias de los códigos fuentes

    - Uno se fija en las dependencias de los códigos fuente debido a que si un cambio en el módulo del código fuente, los otros módulos pueden necesitar ser cambiados o recompilados, y

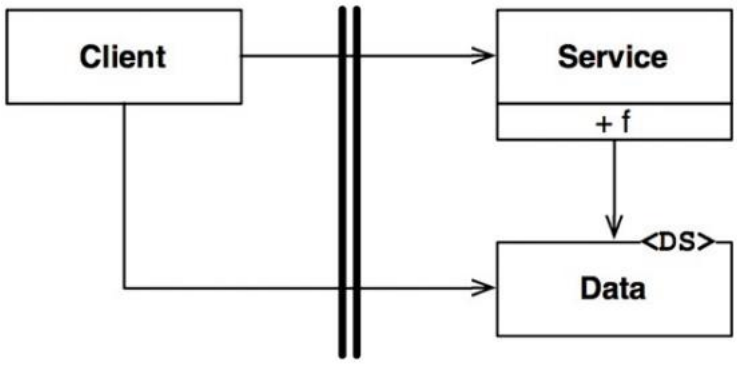
después desplegados. Gestionar y crear *firewalls* contra este cambio es la razón principal de las fronteras

- Las fronteras arquitectónicas más simples y comunes no tienen representación física, sino que es una segregación disciplinada de funciones y datos dentro de un solo procesador y un solo espacio de direcciones (disociación a nivel de código fuente)

  - Desde un punto de vista del despliegue, esto es solo un archivo ejecutable llamado monolito. El hecho de que las fronteras no sean visibles durante el despliegue de un monolito no significa que no sean importantes: la habilidad para desarrollar independientemente varios componentes para el ensamblaje final es muy valioso

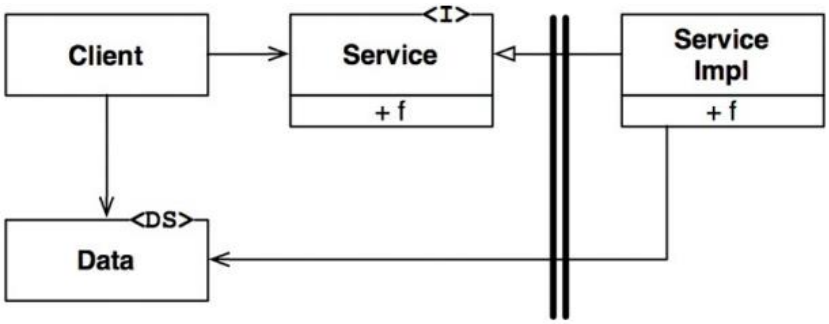
  - Tales arquitecturas siempre dependen de un polimorfismo dinámico para gestionar sus dependencias internas, por lo que el desarrollo orientado a objetos ha sido muy importante estas décadas. Sin OO, los arquitectos deben volver a la práctica de usar punteros a funciones y otras prácticas arriesgadas

  - El cruce más simple es el de una llamada de una función por un cliente de bajo nivel a un servicio de alto nivel, y la dependencia tanto del tiempo de ejecución como de compilación apunta en la misma dirección, al componente de mayor nivel. El cliente llama a la función del servicio y pasa una instancia de los datos a través de un argumento de la función o por otros métodos más elaborados



  - En el caso contrario en el que el cliente de alto nivel necesita invocar un servicio de bajo nivel, el polimorfismo dinámico se usa para invertir la dependencia contra el flujo de control, haciendo que la dependencia del tiempo de ejecución sea opuesta a la dependencia del tiempo de compilación. El cliente de alto nivel llama a la función del servicio de bajo nivel a través de una interfaz, y la dependencia sigue siendo de bajo nivel a alto nivel y la

estructura de datos ahora está en el componente de nivel más alto



- Con este tipo de separación se puede apoyar el desarrollo, la comprobación y el despliegue: los equipos pueden trabajar independientemente en cada uno de sus componentes sin afectar a los otros, haciendo que los componentes de alto nivel no dependan de los de bajo nivel. Las comunicaciones entre componentes son muy rápidas y baratas en esta separación porque en su mayoría son llamadas a funciones

- La representación física más simple de una frontera arquitectónica es una librería dinámica. El despliegue no involucra compilación, sino que los componentes se entregan en binario o en alguna forma desplegable (disociación a nivel de despliegue)

  - Los componentes a nivel de despliegue son lo mismo que los monolitos. Las funciones generalmente existen en el mismo procesador y el mismo espacio de direcciones, y las estrategias para segregar los componentes y gestionar sus dependencias son las mismas

  - Igual que con los monolitos, las comunicaciones a través de las fronteras de componentes desplegables son solo llamadas de funciones, y, por tanto, baratas. Puede haber un *hit* de una sola vez para una vinculación dinámica o una carga en tiempo de ejecución, pero las comunicaciones siguen siendo simples y baratas

  - Tanto los monolitos como los componentes de despliegue pueden usar los *threads*, los cuales no son fronteras o unidades de despliegue, sino que son una manera de organizar y ordenar la ejecución. Pueden estar contenidos enteramente en un componente o pertenecer a varios

- Una frontera arquitectónica más fuerte es el proceso local, que se crea típicamente desde la línea de comandos o una llamada del sistema equivalente

- Los procesos locales se ejecutan en el mismo procesador, o en el mismo conjunto de procesadores dentro de un multinúcleo, pero se ejecutan en espacios de direcciones separados. La protección de memoria generalmente protege estos procesos de compartir memoria, aunque las particiones de memoria compartidas se usan bastante

- Los procesos locales se comunican entre ellos usando *sockets* o algún tipo de facilidad de comunicaciones de sistema operativos tales como correos o colas de mensajes

- Cada proceso local puede ser un monolito vinculado estáticamente o puede estar compuesto de componentes de despliegue dinámicamente vinculados. En el primer caso, varios procesos monolíticos pueden tener los mismos componentes compilados y vinculados en ellos, y en el segundo caso, pueden compartir los mismos componentes de despliegue dinámicamente vinculados

- La estrategia de segregación entre los procesos locales es lo mismo que para los monolitos y los componentes binarios. Las dependencias de código fuente apuntan en la misma dirección a través de la frontera, y siempre apuntan al componente de más alto nivel

- Para los procesos locales, esto significa que los procesos de mayor nivel no pueden contener nombres o direcciones físicas o llaves de registro a procesos de menor nivel

- La comunicación a través de las fronteras de los procesos locales implica llamadas al sistema operativo, ordenación y decodificación de datos y cambios de contexto entre procesos, que son moderadamente costosos

- La frontera más fuerte es un servicio, el cual es un proceso generalmente iniciado desde la línea de comandos o a través de una llamada del sistema equivalente

    - Los servicios no dependen de la localización física: dos servicios comunicados pueden o no operar en el mismo procesador. Los servicios asumen que todas las comunicaciones ocurren en la red

    - Las comunicaciones a través de las fronteras de servicio son muy lentas comparadas con las llamadas de funciones, y la comunicación a este nivel debe lidiar con grandes niveles de latencia

- De otro modo, las mismas reglas de los procesos locales aplican a los servicios: los servicios de menor nivel deben hacer *plugin* a servicios de alto nivel. El código fuente de los servicios de alto nivel no deben tener conocimiento físico de ningún servicio de bajo nivel