Clean Architecture - C6 & C7

Programación Funcional

Principio de Responsabilidad Única

Capítulo 6: Programación Funcional

Programación Funcional

Caso: Imprimir los cuadrados de los primeros 25 números enteros

En Java (orientado a objetos):

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

Usa una variable **mutable**: Una variable que cambia su estado en la ejecución del programa

En Clojure (funcional):

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

```clojure
(println ;_____________________ Imprimir
(take 25 ;_____________________ los primeros 25
(map (fn [x] (* x x)) ;__ cuadrados
(range)))) ;_____________ de los enteros
```

No usa variables mutables: La variable x es inicializada, pero nunca modificada

Esto nos lleva a una **afirmación**: Las variables en los lenguajes funcionales no varían.

Inmutabilidad y arquitectura

¿Por qué un arquitecto se preocuparía por la mutabilidad de las variables?

- Porque todas los problemas de **condicion de secuencia**, **condicion de bloqueo** y **actualizaciones concurrentes** son debido a variables mutables.

-> Todos los problemas que surgen en aplicaciones concurrentes, que requieren múltiples hilos y procesos, no ocurrirían si no hay variables mutables.

Desde el punto de vista de arquitecto es importante, éste quiere diseñar sistemas robustos que soporten múltiples hilos y procesos.

La pregunta sería: ¿Se puede mantener siempre la inmutabilidad?

- Si, pero solo con **infinito almacenamiento y velocidad de procesamiento.**

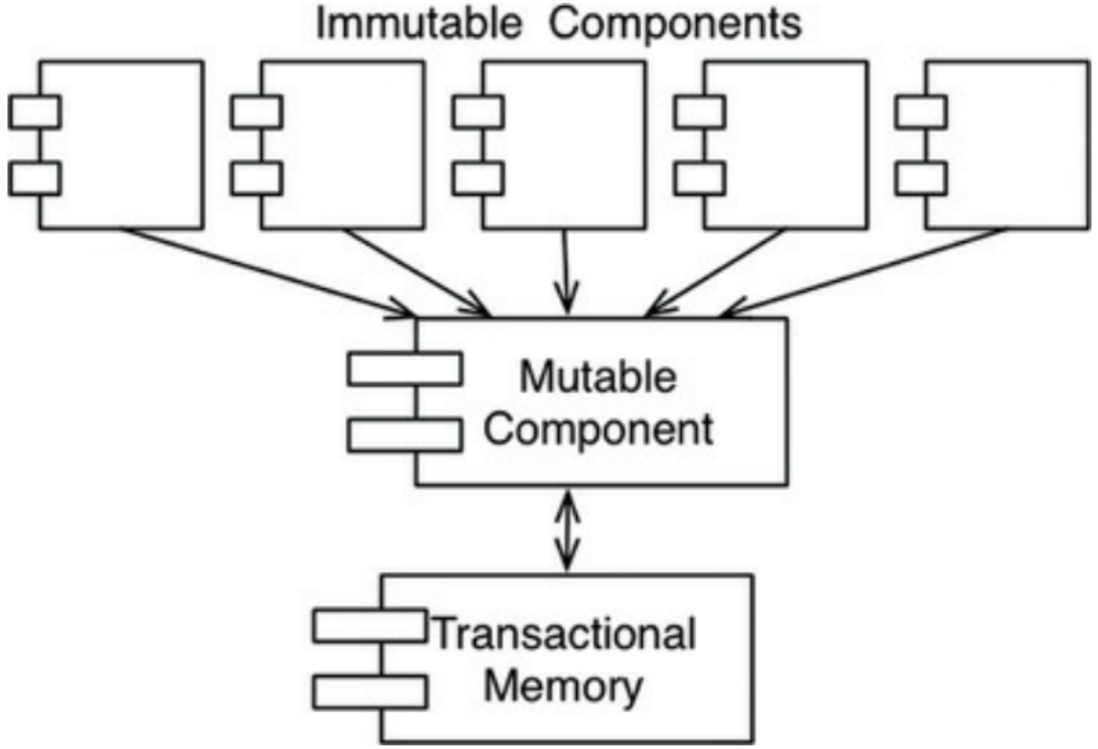
Segregación de mutabilidad

Se refiere a separar los servicios dentro de la aplicación en componentes mutables e inmutables.

Los componentes inmutables realizan sus tareas de forma funcional, sin utilizar mutaciones. Estos se comunican con uno o más componentes que son mutables, y que permiten que el estado de las variables cambie.

Es práctica común utilizar una memoria transaccional para proteger las variables mutables de las actualizaciones concurrentes y las condiciones de secuencia.

La memoria transaccional protege a las variables con un un esquema basado en transacciones o reintentos, de igual manera que las BD.



Suministro de eventos

El aprovisionamiento de eventos es una estrategia en la que almacenamos las transacciones, pero no el estado.

Ejemplo: Una aplicación bancaria que, en lugar de almacenar en lugar de almacenar los saldos de las cuentas, y realizar mutaciones en base a depósitos y retiros, almacenamos sólo las transacciones.

Como mejora se puede calcular y guardar el estado cada medianoche, para evitar gran procesamiento y almacenamiento. Para calcular el estado actual, solo se aplican las transacciones desde la medianoche.

En un caso ideal, si tenemos suficiente almacenamiento y procesamiento, podemos hacer aplicaciones completamente inmutables y, por lo tanto, completamente funcionales.

Conclusión

- La programación estructurada es una disciplina que se opone a la transferencia directa de control.
- La programación orientada a objetos es la disciplina que se opone a la transferencia indirecta de control.
- La programación funcional es la disciplina que se opone a la asignación de variables.

Cada uno restringe algún aspecto de la forma en que escribimos el código. Ninguno de ellos ha aumentado nuestro poder o nuestras capacidades.

Las reglas del software son las mismas hoy que en 1946, cuando Alan Turing escribió el primer código que se ejecutó en un ordenador electrónico. Las herramientas han cambiado y el hardware también, pero la esencia del software sigue siendo la misma.

Parte 3: Principios de diseño

Principios de diseño

SOLID: Cómo organizar funciones y estructuras de datos en clases, y cómo esas clases deben estar conectadas.

El objetivo de estos principios es crear estructuras de software que sean:

- Tolerantes al cambio
- Fáciles de entender
- Sean la base de componentes que puedan utilizarse en muchos sistemas de software.

Estos son:

- El principio de responsabilidad única (SRP)
- El principio de apertura y cierre (OCP)
- El principio de sustitución de Liskov (LSP)
- El principio de segregación de interfaces (ISP)
- El Principio de inversión de dependencias (DIP)

Capítulo 7: Principio de responsabilidad única (SRP)

Principio de responsabilidad única

Es el principio menos entendido de los 5.

Históricamente fue descrito:

> **Un módulo debe tener una, y sólo una, razón para cambiar.**

Interpretando que, por razón se refiere a alguna parte interesada, se puede reformular a:

> **Un módulo debe ser responsable ante un, y sólo un, usuario o parte interesada.**

Agrupamos a los usuarios o stakeholders en una entidad llamada actor. Finalmente:

> **Un módulo debe ser responsable ante un, y sólo un, actor.**

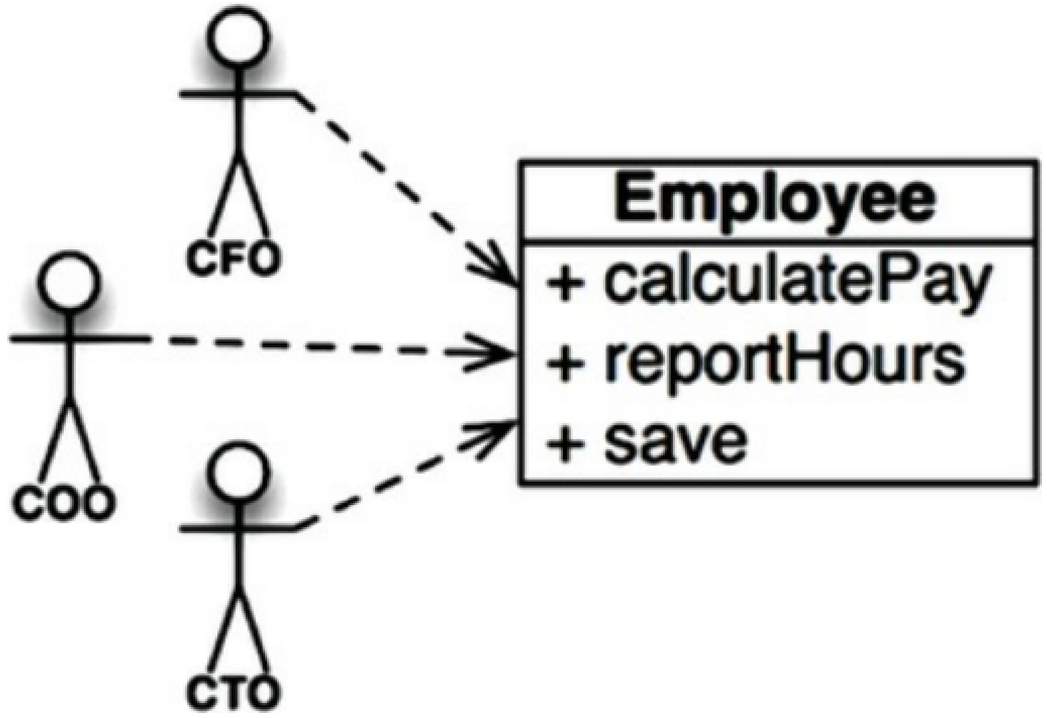
Existen síntomas que se generan cuando este principio no se respeta:

Síntoma 1: Duplicación accidental

Caso: Clase empleado de un sistema de nóminas, tiene 3 métodos o funciones:

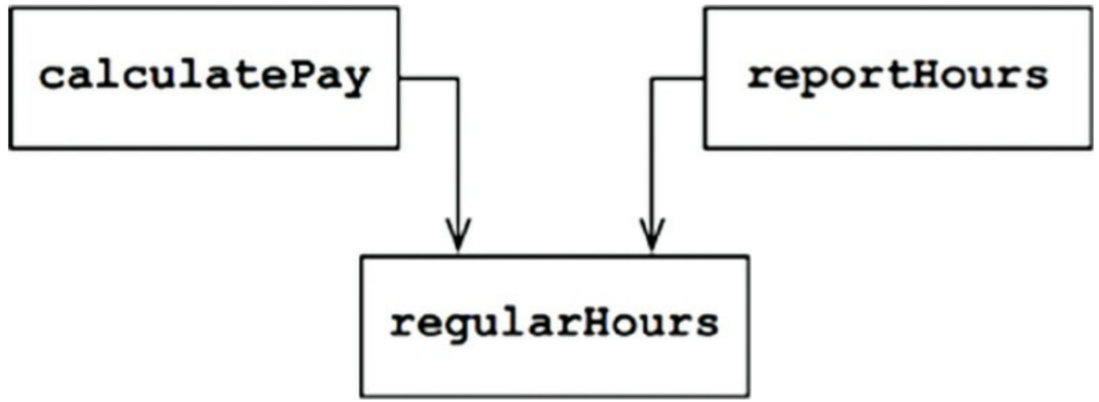
- calculatePay(): Definido por el departamento de contabilidad, que se reporta al CFO.
- reportHours(): Definido y usado por el departamento de RRHH, que se reporta al COO.
- save(): Definido por el administrador de base de datos (DBA), que reporta al CTO.

Esto puede generar que las acciones del equipo del CFO pueden afectar algunas de las que el equipo del COO depende.



Síntoma 1: Duplicación accidental

1. Supongamos que calculatePay() y reportHours() dependen de una función regularHours() para calcular las horas regulares, esto para evitar duplicar código.
2. El equipo del CFO decide cambiar regularHours(), pero el equipo del COO no está al tanto de esto.
3. Un desarrollador hace el cambio sin percatarse de la dependencia adicional de reportHours().
4. Se prueba y se despliega. RRHH sigue ejecutando sus operaciones pero con números erróneos.
5. Finalmente, los errores han costado miles o hasta millones de dólares al presupuesto de RRHH.



Síntoma 2: Uniones o fusiones (merge)

1. Supongamos que el equipo de DBAs del CTO decide que debe haber un cambio de esquema en la tabla Empleados de la base de datos.
2. Supongamos también que el equipo de equipo del COO decide que necesita un cambio en el formato del informe de horas.
3. Dos desarrolladores diferentes, de equipos diferentes, cambian la clase Empleado.
4. Desgraciadamente sus cambios chocan. El resultado es una fusión o merge.
5. Esta fusión pone en riesgo tanto al CTO como al COO, es posible que al COO también.

La forma de evitar este problema es separar el código que soporta diferentes actores.

Solucion 1

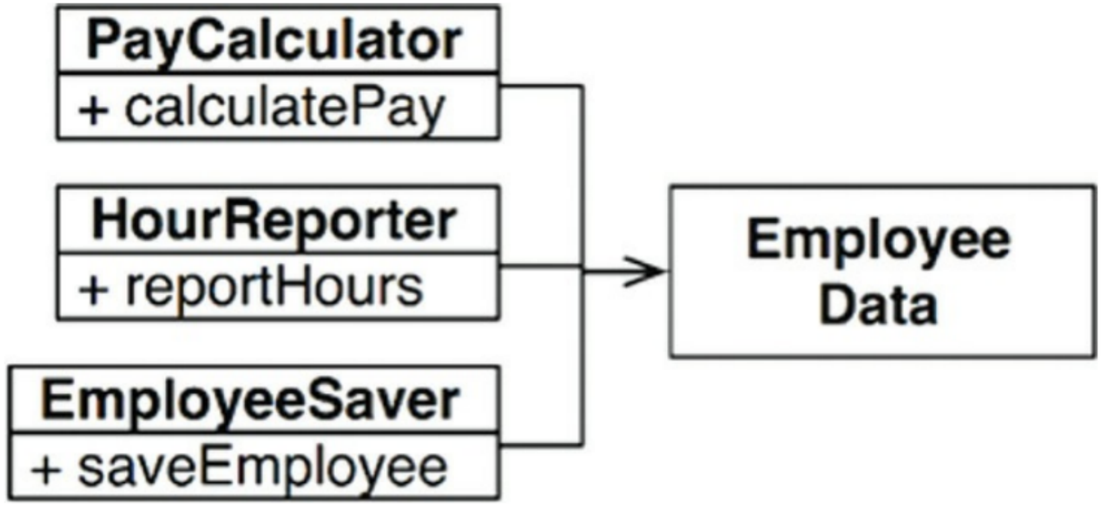
Separa los datos de las funciones.

Las tres clases comparten el acceso a EmployeeData, que es una simple estructura de datos sin métodos.

Cada clase contiene sólo el código fuente necesario para su función particular. Las tres clases no pueden conocerse entre sí.

Así se evita cualquier duplicación accidental.

El inconveniente es que los desarrolladores tienen ahora tres clases que tienen que seguir.

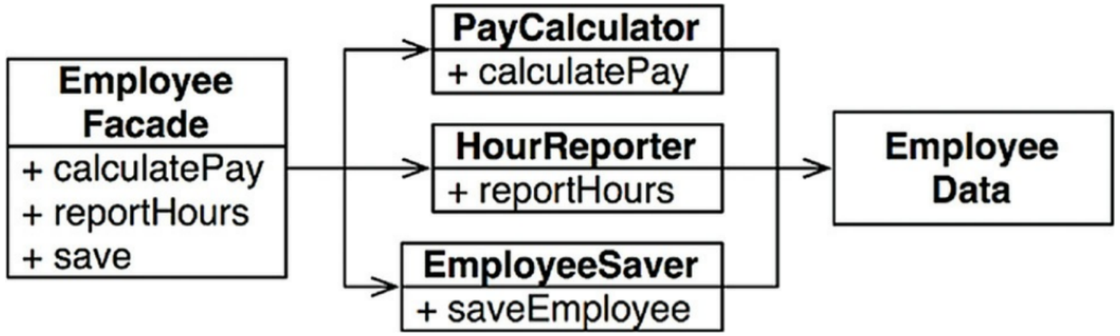


Solucion 1 v2

Una solución común es utilizar el patrón de diseño **Facade**.

La clase EmployeeFacade contiene muy poco código. Es responsable de instanciar y delegar en las clases con las funciones.

De igual manera, algunos desarrolladores prefieren mantener las reglas de negocio más importantes más cerca de los datos.



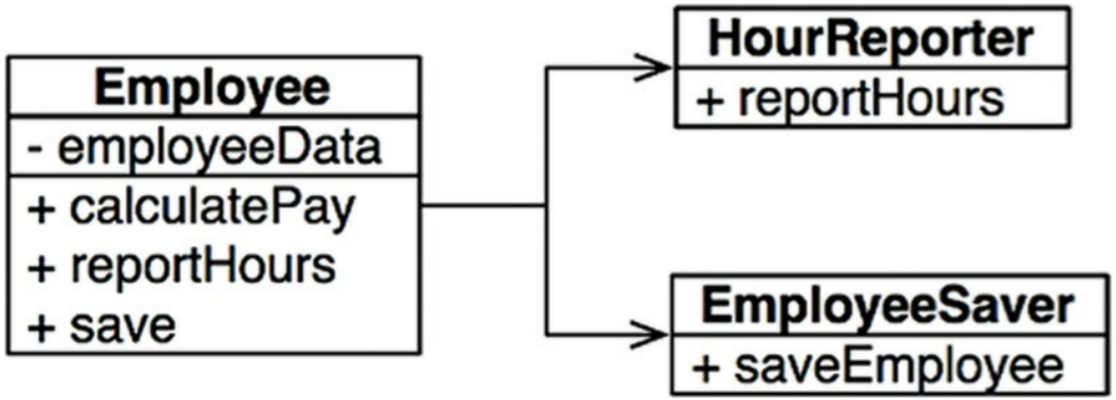
Solucion 2

Esto puede hacerse manteniendo el método más importante en la clase original Empleado y luego usar esa clase como un **Facade** para las funciones menores.

¿Se podría contradecir a estas soluciones considerando que cada clase debe contart con una sola función?. **Difícilmente.**

El número de funciones necesarias para calcular la paga, generar un informe, o guardar los datos es probable que sea grande en cada caso. Cada una de esas clases (consideradas un **ámbito**) tendría muchos métodos privados en ellas.

Fuera de ese fuera de ese ámbito, nadie sabe que los miembros privados de la familia existen.



Conclusión

- El Principio de Responsabilidad Única se refiere a las funciones y las clases, pero reaparece de forma diferente en otros dos niveles.
- A nivel de componentes, se convierte en el *Principio de Cierre Común*.
- En el nivel arquitectónico, se convierte en el *Eje de Cambio responsable de la creación de Límites Arquitectónicos*, conceptos que se verán en los siguientes capítulos.

Referencias

- Libro de Clean Architecture
- Resaltador de código
- Condición de secuencia
- Condición de bloqueo
- Actualizaciones concurrentes

Muchas Gracias

Correo · LinkedIn · Github