ORTOGONALIDAD Y DEPENDENCIAS

Ortogonalidad

Dos componentes A y B son ortogonales ($A \perp B$) si los cambios en uno no afectan al otro. Suponen más independencia y menos acoplamiento

Ejemplos

- La base de datos debe ser ortogonal a la interfaz de usuario
- En un helicóptero, los mandos de control no suelen ser ortogonales

El Arte del Equilibrio: Cómo se Pilota un Helicóptero

Los Mandos de Vuelo Básicos

El Cíclico (Mano Derecha)

Se mueve en cualquier dirección para desplazar el helicóptero hacia ese mismo lado.

El Colectivo y Acelerador (Mano Izquierda)

El colectivo genera sustentación sublendo las palas, mientras el acelerador regula la potencia.

Pedales (Pies)

Varían el empuje del rotor de cola para controlar el giro o dirección de la nariz.

El Desafío de la Interdependencia

El Efecto Secundario Constante

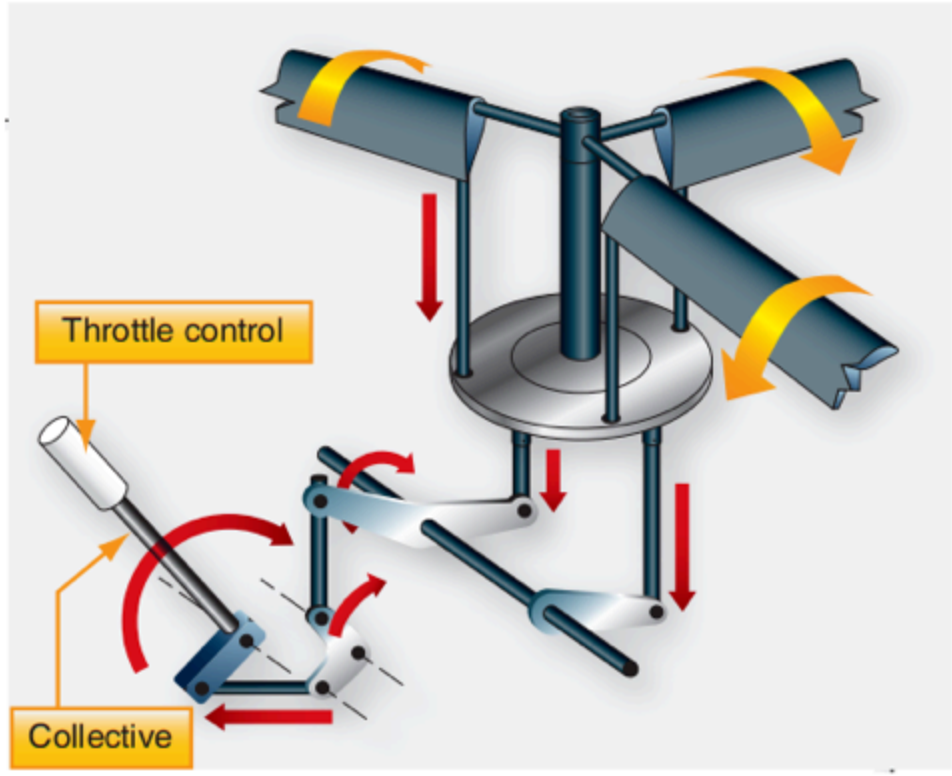
Cada acción en un mando afecta inevitablemente la estabilidad de los otros tres.

Reacción en Cadena al Descender

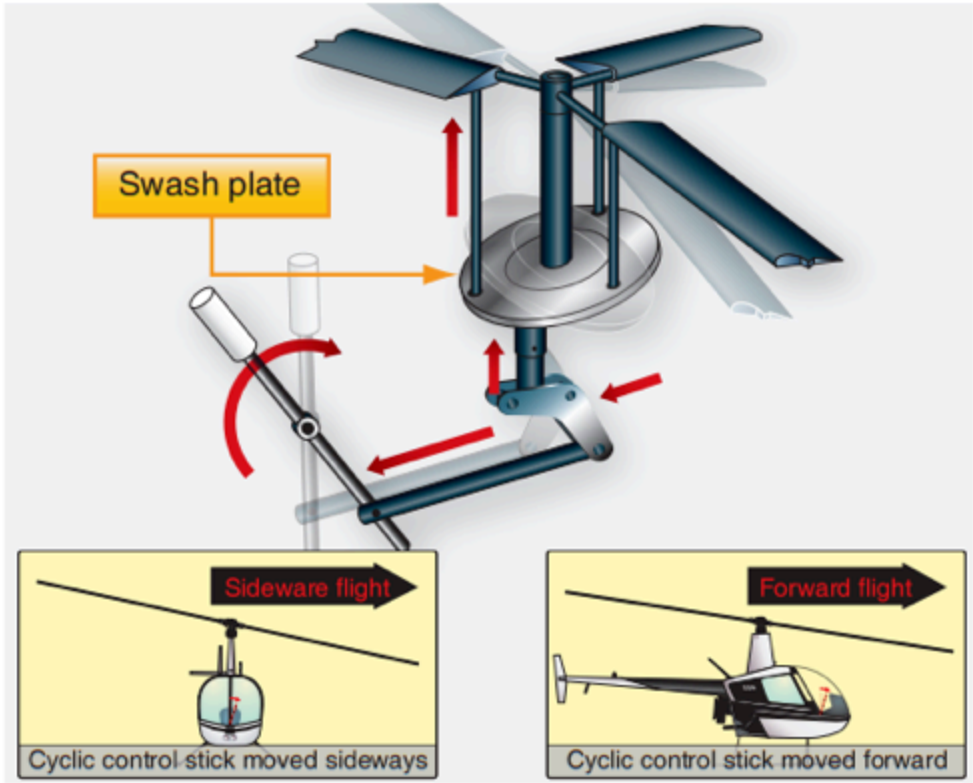
Bajar el colectivo exige compensar con el ciclico hacia atrás y presionar el pedal derecho.

Carga de Trabajo Fenomenal

Manos y pies deben moverse sin pausa para equillbrar las fuerzas que interactúan.



El colectivo modifica el paso de todas las palas del rotor simultáneamente y en la misma medida, aumentando o disminuyendo así la sustentación.



El cíclico modifica el ángulo del plato cíclico, lo que cambia el plano de rotación de las palas del rotor. De este modo, la aeronave se desplaza horizontalmente en cualquier dirección en función de la posición del cíclico.

Beneficios de la ortogonalidad

Mayor productividad

- Es más fácil escribir un componente pequeño y auto-contenido que un bloque muy grande de código.
- El tiempo de desarrollo y **pruebas** se reduce
- Se pueden combinar unos componentes con otros más fácilmente. Mayor **reutilización**.
- En teoría, si $A \perp B$, el componente A sirve para m propósitos y B sirve para n, entonces $A \cup B$ sirve para $m \times n$ propósitos.
- La falta de cohesión perjudica la reutilización $\longrightarrow$ ¿y si hay que hacer una nueva versión gráfica de una aplicación de línea de comandos que lleva incrustada la escritura en consola con `System.out.println`? ¡Puede descohesionar!

Menor riesgo

- Defectos aislados, más fáciles de arreglar
- Menor **fragilidad** del sistema global. Los problemas provocados por cambios en un área se limitan a ese área
- Más fácil de **probar**, pues será más fácil construir pruebas individuales de cada uno de sus componentes (por ejemplo, las técnicas de *mocking* son más sencillas)

Aplicación de la ortogonalizad

La ortogonalidad es aplicable a:

- el diseño
- la implementación
- las pruebas
- bibliotecas
- la documentación

A nivel de *diseño*, los patrones de diseño y las arquitecturas como MVC facilitan la construcción de componentes ortogonales.

Lectura recomendada

- Leer el Topic 10: Orthogonality de D. Thomas & A. Hunt. The Pragmatic Programmer: your journey to mastery, 2nd Edition, Addison-Wesley, 2020.

Técnicas de implementación

Técnicas de implementación para fomentar la ortogonalidad:

- Hacer **refactoring**
- Codificar **patrones** de diseño: strategy, template method, etc.
- Evitar datos globales y **singletons**: ¿qué pasaría si hubiera que hacer una versión *multithreaded* de una aplicación?
- **Inyectar**: pasar explícitamente el contexto (dependencia) como parámetro a los constructores
- Usar **anotaciones** (Java), decoradores (TypeScript) o atributos (C#)
- **Desacoplar**: Ley de *Demeter* — «No hables con extraños»
- Usar programación orientada a **aspectos**

Desacoplar - ley de Demeter

Al pedir un servicio a un objeto, el servicio debe ser realizado de parte nuestra, no que nos devuelva un tercero con el que tratar para realizarlo

Ejemplo

```
public boolean canWrite(User user) {
  if (user.isAnonymous())
    return false;
  else {
    return user.getGroup().hasPermission(Permission.WRITE);
  }
}
```

Refactorización: definir un método `User.hasPermission()`

Lectura recomendada

- Leer el Topic 28: Decoupling de D. Thomas & A. Hunt. The Pragmatic Programmer: your journey to mastery, 20th Anniversary Edition, 2nd Edition, Addison-Wesley Professional, 2020.

Ley de Demeter para funciones

Los métodos de un objeto solo deben hacer llamadas a métodos...

1. **propios**
2. de objetos pasados como **parámetros**
3. de objetos **creados** por ellos mismos
4. de objetos **declarados** en el mismo método

```
class Demeter {
  private A a;
  private int func();
  public void example (B b);

  void example(B b) {
    C c;
    int f = func();  // (caso 1)
    b.invert();      // (caso 2)
    a = new A();
    a.setActive();   // (caso 3)
    c.print();       // (caso 4)
}
```

Excepción: Interfaces *fluent*

Hay una excepción notable a la prohibición de encadenar llamadas a funciones de la ley de Demeter. Esta regla no aplica si es muy poco probable que haya cambios en las cosas que se encadenan.

En la práctica, cualquier parte de tu aplicación debe considerarse como algo que es probable que cambie; cualquier elemento de una biblioteca de un tercero debe considerarse volátil, en particular si quienes mantienen dicha biblioteca suelen cambiar su API de una versión a otra.

Ejemplos de código como el siguiente son aceptables como excepción a esta interpretación de la ley de Demeter.

¿Por qué?

```
java.util.List<String> myList =
    Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

Los métodos `stream`, `filter`, `map`, `sorted` y `forEach` son parte de las nuevas *interfaces funcionales* de Java para manejar *streams*, incorporadas a las colecciones (v.g. `List`) desde la versión Java 8.

Este tipo de interfaces como la del API de streams de Java se conoce como *fluent interfaces*.

Críticas a la ley de Demeter

La ley de Demeter, ¿realmente ayuda a crear código más mantenible?

Ejemplo: pintar gráficos de grabadoras

- Pintar un gráfico con los datos registrados por una serie de grabadoras (`Recorder`) dispersas por el mundo.
- Cada grabadora está en una ubicación (`Location`), que tiene una zona horaria (`TimeZone`).
- Los usuarios seleccionan (`Selection`) una grabadora y pintan sus datos etiquetados con la zona horaria correcta...

```
public void plotDate(Date aDate, Selection aSelection) {
  TimeZone tz = aSelection.getRecorder().getLocation().getZone();
}
```

Críticas

- Multiplicidad de dependencias: `plotDate` ⟶ `Selection`, `Recorder`, `Location`, `TimeZone`.
- Si cambia la implementación de `Location` de forma que ya no incluye directamente una `TimeZone`, hay que cambiar `plotDate`
- Añadir un método *delegado* `getTimeZone` a `Selection`. Así `plotDate` no se entera de si la `TimeZone` le llega desde `Recorder` o desde un objeto contenido en `Recorder`.

```
public void plotDate(Date aDate, TimeZone tz) {
  /* ... */
}
plotDate(someDate, someSelection.getTimeZone());
```

Ahora `plotDate` ⟶ `Selection`, `TimeZone`, pero se han eliminado las restantes dependencias.

- Costes de espacio y ejecución de métodos *wrapper* que reenvían la petición al objeto delegado: violar la ley de Demeter para mejorar el **rendimiento**
- Otros ejemplos de mejora del rendimiento: desnormalización de BBDD

Ortogonalidad en toolkits y bibliotecas

Muchas bibliotecas actuales implementan la ortogonalidad a través de metadatos, o atributos o etiquetas (`@tag`), también llamados **anotaciones** en Java y **decoradores** en TypeScript.

Los metadatos se emplean para proporcionar propósitos específicos, como v.g. persistencia de objetos, transacciones, etc. Por ejemplo, Spring o EJB utilizan anotaciones @ declarativas para expresar la transaccionalidad de una operación o la persistencia de una propiedad de una clase fuera del método que debe ejecutar dichas funcionalidades.

Otro método para implementar la ortogonalidad es usar Aspectos y *Aspect-Oriented Programming* (AOP). Este método es empleado por el framework Spring.