Biblioteca vs. Framework

Versión 0.3 - Nov 2020

Índice:

Versiones

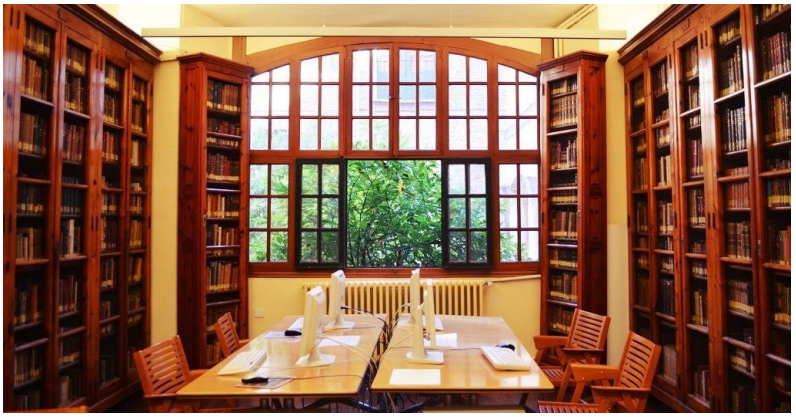
Autores principales	Versión	Fecha	Observaciones
Nicolás Anderson	0.3	2020	Cambios de sintaxis menores. Cambio de ejemplos bibliotecas.
Rodolfo Caputo, Franco Bulgarelli	0.2	2016	Versión original

Introducción

¿Biblioteca? ¿Framework? ¿Qué son esas cosas?

A medida que uno comienza a desarrollar más y más grandes piezas de software, es común empezar a ver y utilizar nuevas herramientas en tal proceso, y a veces uno simplemente las usa por inercia, como entiende que se usan, y las nombra como oye que se nombran, sin entender bien del todo qué es lo que está usando.

Ese puede ser el caso de las bibliotecas y los frameworks, dos herramientas casi omnipresentes en el desarrollo industrial de software, que tienen muchos puntos en común pero una serie de diferencias en su concepción y su uso que a veces no quedan completamente claras.



La necesidad de reutilizar

Para entender un poco de qué estamos hablando, remontémonos a los orígenes del desarrollo de software:

Originalmente, toda funcionalidad que fuera necesaria para una aplicación, había de ser implementada a mano por el equipo encargado de la construcción del producto en cuestión. Por ejemplo, si se necesitaba conocer la longitud de un string, podía implementarse una función del estilo:

```
int string_length(char* unString){
        int i = 0, longitud = 0;
        while(unString[i] != '\0'){
                longitud++, i++;
        }
        return longitud;
}
```
[1]

[1] La implementación real de esta función es mucho, mucho más concisa -y menos clara-.

Y luego utilizarla:

```
char* palabra = “Una palabra”;
printf("La palabra es %s y tiene %d caracteres", palabra,string_length(palabra));
```

Y reutilizarse múltiples veces dentro del proyecto, evitando así la necesidad de desarrollar funcionalidad ya existente, evitando repetir lógica y generando una abstracción nueva.

Sin embargo, eventualmente el proyecto finalizaría, uno nuevo arrancaría y las probabilidades indican que seguramente la función *string_length* sería necesaria nuevamente y debería ser desarrollada una vez más, prácticamente *replicando* el código de otro proyecto anterior.

Fue entonces que surgieron las bibliotecas, como forma de compartir funciones y estructuras ya desarrolladas entre varios proyectos.

Por ejemplo, en C tenemos una biblioteca estándar que nos provee varias funciones para trabajar con strings, como nuestra *string_length* (*strlen* en C) bajo el header *string.h*. O la biblioteca Commons Lang para Java que también nos brinda, entre otras funcionalidades, métodos para manejar o manipular strings

Uno simplemente añadiendo dicha biblioteca al proyecto puede utilizar la clase StringUtils para, y por nombrar algunos ejemplos, obtener la diferencia de caracteres entre dos strings con el método *difference* o contar la cantidad de veces que aparece un caracter en un string con el método *countMatches,* porque alguien más ya la hizo.

Cabe destacar, asimismo, que no sólo existen bibliotecas genéricas, como aquellas para manejar strings o fechas, que son reutilizadas en varios proyectos sino también otras específicas de un sistema, un dominio en particular, como aquella que gestione los permisos internos dentro de una aplicación, que puede ser utilizada múltiples veces dentro del mismo sistema, aunque difícilmente traspase sus límites.

La biblioteca se desarrolla, se compila y se carga de forma separada[2] al código de nuestra aplicación, pero las funcionalidades que define son utilizadas indistintamente por nuestro código como si de parte de él se tratase.

Las bibliotecas resuelven un problema de reutilización de lógica asociada a abstracciones, representada e implementada a través de código.

[2] Para más información sobre bibliotecas compartidas ver *Fundamentos de Sistemas Operativos*, Abraham Silberschatz, Sección 8.1.5.

Cuando las bibliotecas no alcanzan

Si bien las bibliotecas nos permiten reutilizar grandes cantidades de código existente, no resuelven todos nuestros problemas de reutilización. Veamos el siguiente ejemplo:

Un desarrollador hace un proyecto, y a su vez, para poder garantizar la calidad del mismo, decide automatizar algunas pruebas sobre su funcionamiento.

Entonces empieza a armar su código:

```
void main(String[] args){
	if (args[1] == "run-tests"){
		correrTests()
	}
	else{
		ejecutarPrograma(args)
	}
}
```

```
void ejecutarPrograma(String[] args){ … }


void correrTests(){

	test1();
	test2();
	test3();
	...
}

void test1(){
	prepararTests()
	// (...) código del test1
	limpiarTodo()
}

void test2(){
	prepararTests()
	// (...) código del test2
	limpiarTodo()
}
```

Y así arma su proyecto con sus tests, y es feliz.

Al mes siguiente, un nuevo proyecto llega y se ve en la necesidad de armar un proyecto similar, para permitir el testeo. Entonces, aprovechando que le resultó, utiliza el mismo esquema:

- En el *main*, si le pasan “run-tests” corre los tests.
- Si no le pasan “run-tests” ejecuta el programa.
- En *ejecutarPrograma* escribe el código posta de la aplicación.
- En la función *correrTests* define los tests que va a correr.
- En cada test, llama a *prepararTests* al inicio y a *limpiarTodo* al final.
- En *prepararTests* define el fixture de datos a usar en todos los tests.
- En *limpiarTodo* libera recursos que pueda haber necesitado tomar en el test.

Pasan los meses y otros proyectos llegan, los cuales son encarados con el mismo esquema. Nuevas personas participan de los proyectos y se les enseña a desarrollar “siguiendo ese esquema”.

El “esquema”, dado su éxito funcional, se sigue utilizando. Pero no es esta la única razón de su utilización. El “esquema” provee algo fundamental: Una estructura, una forma de trabajar ya definida, que evita que el desarrollador vuelva a pensar cómo estructurar determinadas tareas, simplemente descansa en algo que ya alguien pensó y sigue esa forma de hacer las cosas.

El esquema es reutilizado múltiples veces a lo largo de proyectos, al igual que sucede con las bibliotecas. El esquema representa abstracciones (*correrTests*, *prepararTest*, etc.) con una lógica común, al igual que las bibliotecas.

Sin embargo, hasta el momento, las bibliotecas podían reutilizarse casi sin esfuerzo alguno, mientras que el “esquema” es solo una lista de conceptos que hay que implementar una y otra vez. Pero… ¿Por qué? Si, además de reutilizar nuestra lógica de cada dominio, queremos aprovecharlo para reutilizar el control de flujo que nos provee, ¿no es posible generar código reutilizable a partir de este esquema?

Sí, lo es. Y por eso nuestro desarrollador, cansado de implementar decenas de veces su famoso “esquema” se lanza a codificar una primera versión de código que lo ayude a aplicar más rápida y fácilmente esta estructura en sus proyectos:

```
class AplicacionConTests(){
	void main(String[] args){
		if (args[1] == "run-tests"){
			correrTests()
		}
		else{
			ejecutarPrograma(args)
		}
	}
```

```
        abstract List< (void -> void) > = tests

        void correrTests(){
                tests.foreach( test => correrTest(test))
        }

        void correrTest((void->void) test){
                prepararTest()
                test.run()
                limpiarTodo()
        }

        abstract void prepararTest()
        abstract void limpiarTest()
        abstract void ejecutarPrograma(String[] args)
}
```

Y luego define su programa:

```
class MiApp extends AplicacionConTests{

        void test1(){
                //codigo del test1
        }

        void test2(){
                //codigo del test2
        }

        override tests = new List(test1,test2)

        override def prepararTest(){ // crear fixture… }

        override def limpiarTodo(){ // liberar recursos...}

        override def ejecutarPrograma(){ // código del programa posta...}

}
```

Y cada vez que necesita empezar un nuevo proyecto no es necesario implementar toda la estructura nuevamente, sino que es suficiente para cualquiera extender la clase *AplicaciónConTests* y redefinir los métodos *tests*, *ejecutarPrograma*, *limpiarTodo* y *prepararTest* que son aquellos cuyo comportamiento depende puramente del dominio en cuestión.

Henos aquí con una nueva forma de reutilizar código: no solo se definen funciones y componentes que pueden ser reutilizados, sino que se definen piezas de software

reutilizables que permiten ser extendidas[3] para cubrir las necesidades particulares de cada proyecto pero a su vez definen una forma, un marco de trabajo común para todos ellos.

Es por eso último que a estos componentes de software se los ha dado en llamar "*Frameworks*".

Los frameworks definen una estructura, una forma de trabajar siguiendo determinados lineamientos.

Bajando a lo concreto

Las bibliotecas definen funcionalidades y estructuras concretas que pueden ser utilizadas o no por el desarrollador en un proyecto que las incluye, de la forma que el desarrollador considere oportuna.

Por ejemplo: un desarrollador puede elegir utilizar LocalDate o utilizar otra clase propia que le convenga más, por ejemplo porque ordena mes y año como se hace en español)

El control del flujo del programa es manejado por el usuario que eventualmente instancia estructuras de la biblioteca o hace llamadas a operaciones que ésta define. Es decir, cada llamada hace algún trabajo y retorna el control al usuario que la llamó.
(En el ejemplo se ve que es el desarrollador el que define el método main y llama a la función string_length)

Los frameworks definen código que representa las abstracciones y la lógica surgidas de una estructura de trabajo. Los frameworks obligan al desarrollador a trabajar de una forma específica, siguiendo los lineamientos que definen.

Los frameworks condicionan fuertemente el diseño de un componente de software. Suelen manejar ellos el flujo de ejecución del programa e invertir el control, siendo ellos quienes deciden cuándo llamar al código del usuario. *(El desarrollador jamás define el método main, sino que es el framework el que lo define y eventualmente llama al código del usuario, como ejecutarPrograma)*

Es común que los frameworks definan aspectos tales como el orden de ejecución de ciertas tareas, la forma en la que se van a escribir ciertas operaciones (DSLs[4]) o incluso las estructuras de directorios que se van a utilizar para guardar el código fuente.

---

[3] De forma similar a como ocurre en el *template method,* hay una parte básica del código ya definida, que descansa en otra parte del código por ser definida para cada caso.
[4] DSL es la abreviación en inglés de *domain-specific language*. Son lenguajes, generalmente pequeños enfocados en un aspecto particular. Un ejemplo es el lenguaje SQL para realizar consultas en base de datos relacionadas.

También suele ser común que definan clases abstractas que el código cliente extenderá y métodos abstractos que son los que el usuario ha de implementar con el comportamiento propio de la aplicación. *(El framework define la clase abstracta AplicaciónConTests que el usuario extiende completando definiciones de métodos como limpiarTodo y prepararTest)*

En el sitio de Akka HTTP (una suite de bibliotecas para manejo de HTTP sobre Scala, dado que argumentan *no* ser un framework) definen un framework de la siguiente forma:

> *"Un framework, como nos gusta ver el término, te da un "marco", en el cual vos construís tu aplicación. Viene con un montón de decisiones ya pre-tomadas y provee unos cimientos incluyendo estructuras de soporte que te permiten empezar y entregar resultados rápidamente. En cierta forma, un framework es como un esqueleto sobre el cual vos ponés la "carne" de tu aplicación de forma de traerla a la vida. Como tales, los frameworks funcionan mejor si los elegís antes de comenzar el desarrollo de tu aplicación y tratás de apegarte a 'la forma de hacer las cosas' que te define el framework"*

Diferencias y similitudes

Entonces, en definitiva, ¿cuáles son las similitudes y las diferencias entre una biblioteca y un framework?

Una biblioteca y un framework tienen en común que:

Ambos definen abstracciones y lógica propias
Ambos implementan dichas abstracciones y lógica en código reutilizable.

Sin embargo, se diferencian en que:

Bibliotecas	**Frameworks**
Es responsabilidad del desarrollador decidir cómo y cuándo utilizar los componentes	Este define la forma en la que se estructurará el código y el desarrollador se ve obligado a seguir esa forma
Las decisiones de diseño que se toman suelen tener bajo impacto en el diseño del código que la utiliza	Las decisiones de diseño que se toman pueden condicionar fuertemente el diseño del código cliente
Utilizan **control directo**: El usuario llama funciones de la biblioteca e instancia las estructuras que la biblioteca pueda definir.	Utilizan **control inverso**: Es el framework el que llama funciones abstractas que el usuario define en concreto.