Linux Professional Institute

Open Source Essentials

Versión 1.0
Español

050

Table of Contents

Linux Professional Institute

Tema 051: Fundamentos del software

051.1 Componentes del software

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 051.1

Peso

2

Áreas de conocimiento clave

Comprender el concepto de código fuente y ejecución de código Comprender el concepto de compiladores e intérpretes Comprender el concepto de bibliotecas de software

Lista parcial de archivos, términos y utilidades

- Código fuente
- Programas ejecutables
- Código de bytes
- Código de máquina
- Compilador
- Enlazador
- Intérprete
- Máquina virtual en tiempo de ejecución
- Algoritmo
- Bibliotecas de software
- Vinculación estática y dinámica

Linux Professional Institute

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	051 Fundamentos del software
Objectivo:	051.1 Componentes del software
Lección:	1 de 1

Introducción

El *software libre* y el *código abierto* (a menudo conocido como FOSS) se ha convertido en una parte integral de nuestra vida cotidiana, sin que nos demos cuenta. FOSS por ejemplo, se puede encontrar detrás de todas nuestras actividades en Internet en alguna forma: En la computadora donde vemos páginas web en el navegador, o en servidores que almacenan estas páginas web y las entregan tan pronto como las invocamos.

¿Qué es el software?

Sin embargo, antes de analizar todos los detalles del software libre y de código abierto, debemos aclarar qué es realmente el software. Comencemos con una descripción muy general: el software es la parte no física e inmaterial de las computadoras en cualquier forma. El software garantiza que las partes físicas (el *hardware*) de la computadora interactúen y que la computadora pueda aceptar comandos y ejecutar tareas.

Un teléfono inteligente, un ordenador portátil o un servidor en un centro de datos no es más que una máquina hecha de metal y plástico cuando está apagada. Tan pronto como se enciende, el software se inicia, y se forman secuencias de comandos codificadas que controlan los

componentes individuales de esta máquina, permitiendo al usuario interactuar con la computadora y realizando tareas muy específicas llamando a aplicaciones individuales.

Es trabajo de los *desarrolladores de software* analizar las tareas que se supone que debe realizar la computadora y especificarlas de una manera que le permita implementarlas. Las herramientas utilizadas por los desarrolladores son tan numerosas y diversas como las tareas realizadas por el software.

Algunas tareas de software están estrechamente relacionadas con el hardware y la arquitectura de la computadora, por ejemplo, el direccionamiento y gestión de la memoria o el manejo de diferentes procesos. Por lo tanto, los *programadores de sistemas* trabajan cerca del hardware.

Los *desarrolladores de aplicaciones*, por otro lado, se centran más en el usuario y en las aplicaciones de programa que permiten a los usuarios realizar sus tareas de forma eficiente e intuitiva. Un ejemplo de una aplicación compleja es un programa de procesamiento de textos que proporciona todas las funciones para formatear texto en menús o botones y también muestra el texto tal como finalmente se imprimirá.

En general, un *algoritmo* es una forma de resolver un problema. Por ejemplo, para calcular un promedio, el algoritmo normal consiste en sumar una colección de valores y dividir la suma por el número total de valores. Aunque tradicionalmente los algoritmos son diseñados y ejecutados por programadores, hoy en día también los generan la inteligencia artificial.

Los conceptos de este capítulo pueden ayudarle a comprender las fortalezas y los riesgos del FOSS, tomar decisiones informadas e incluso decidir si desea ser desarrollador de software..

Lenguajes de programación

Los *lenguajes de programación* son lenguajes artificiales altamente estructurados que le dicen a una computadora qué hacerla estructura de control que inicia una. Los programas suelen estar escritos en texto, pero algunos idiomas están escritos en forma gráfica. Los desarrolladores de software escriben instrucciones (llamadas *código*) en la computadora en este lenguaje artificial. Sin embargo, el hardware de la computadora no ejecuta directamente este código. El hardware puede ejecutar directamente sólo una serie de patrones de bits almacenados en la memoria, llamados *código de máquina* o *lenguaje de máquina*. Todos los lenguajes de programación son convertidos a código de máquina por un *compilador* o interpretados por otro programa de código de máquina llamado *intérprete* para hacer que el hardware ejecute estas instrucciones.

Algunos de los lenguajes de programación más utilizados actualmente son Python, JavaScript, C, C++, Java, C#, Swift y PHP. Cada uno de estos lenguajes de programación tiene sus propias fortalezas y debilidades, y la elección del lenguaje depende del proyecto y las necesidades del

desarrollador. Por ejemplo, Java es una opción popular para desarrollar aplicaciones empresariales a gran escala, mientras que Python se utiliza a menudo para informática científica y análisis de datos.

Los desarrolladores han demostrado una creatividad impresionante en el diseño de lenguajes de programación. Originalmente eran *lenguajes de bajo nivel* que se parecían a las instrucciones de la computadora. Los lenguajes se han vuelto cada vez más de alto nivel, lo que significa que intentan representar poderosas combinaciones de instrucciones en términos breves. Algunos lenguajes reflejan la forma natural de pensar de las personas, conservando al mismo tiempo el rigor necesario para ejecutarse correctamente.

Actualmente se reconocen alrededor de 400 lenguajes de programación, aunque muchos se utilizan sólo en aplicaciones muy específicas o entornos heredados. Cada uno fue desarrollado con el fin de resolver determinadas tareas.

Características, sintaxis y estructura de los lenguajes de programación.

La elección del lenguaje de programación puede tener un impacto significativo en el rendimiento, la escalabilidad y la facilidad de desarrollo de un proyecto de software. Estas secciones exponen elementos importantes de los idiomas.

Características de los lenguajes de programación

Algunas de las características y cualidades comunes de los lenguajes de programación incluyen:

Concurrencia

La concurrencia denota el manejo de múltiples tareas simultáneamente, ya sea ejecutándolas en diferentes procesadores de hardware o alternando el uso de las tareas de un solo procesador. El grado de concurrencia admitido por un lenguaje de programación puede afectar en gran medida su rendimiento y escalabilidad, especialmente para aplicaciones que requieren procesamiento en tiempo real o grandes cantidades de datos. Cada trabajo por separado podría denominarse *proceso*, *tarea* o *hilo*.

Gestión de memoria

La gestión de memoria es la asignación y liberación de memoria en un programa. Dependiendo del lenguaje de programación o del entorno de ejecución, la gestión de la memoria la puede realizar el programador manualmente o de forma automática. La gestión adecuada de la memoria es crucial para garantizar que un programa la utilice de forma eficaz y que no se quede sin esta ni cause otros problemas. Si un programa no logra liberar la memoria no utilizada, el programa provoca una *pérdida de memoria* que aumenta gradualmente el uso de esta hasta que el programa falla o se notan efectos negativos en el rendimiento.

Memoria compartida

La memoria compartida es un tipo de mecanismo de comunicación entre procesos que permite que múltiples procesos lean y manipulen una región común de la memoria. La memoria compartida es común en hardware como las unidades de disco y también puede ser una forma eficiente de compartir datos entre procesos. Pero el mecanismo requiere una sincronización y gestión cuidadosa para evitar la corrupción de datos. Un error conocido como *condición de carrera* se produce si un proceso realiza un cambio inesperado en los datos mientras otro proceso los está utilizando.

Paso de mensajes

El paso de mensajes es un mecanismo de comunicación entre procesos que les permite intercambiar datos y coordinar sus actividades. Esto se usa comúnmente en programación concurrente para lograr la comunicación entre procesos y se puede implementar a través de varios mecanismos, como sockets, canalizaciones o colas de mensajes.

Recolección de basura

La recolección de basura es una técnica de administración automática de memoria utilizada por algunos lenguajes de programación para recuperar memoria que ya no se usa mientras se ejecuta un proceso. Esto puede ayudar a prevenir pérdidas de memoria y facilitar que los desarrolladores escriban código correcto y eficiente, pero también puede introducir una sobrecarga de rendimiento y dificultar el control sobre el comportamiento preciso del programa.

Tipos de datos

Los tipos de datos determinan qué tipo de información se puede representar en el programa. Los tipos de datos pueden estar predefinidos en el lenguaje o definidos por el usuario, y pueden incluir números enteros, números de punto flotante (es decir, aproximaciones de números reales), cadenas de texto, arreglos y otros.

Entrada y salida (I/O - Input/Output)

La entrada y la salida son mecanismos para leer y escribir datos hacia y desde un programa. La entrada puede provenir de una variedad de fuentes, como clics del usuario y entradas del teclado, un archivo o una conexión de red, mientras que la salida se puede enviar a una variedad de destinos, como una pantalla, un archivo o una conexión de red. La I/O permite que los programas interactúen con el mundo exterior e intercambien información con otros sistemas.

Manejo de errores

El manejo de errores detecta y responde a los errores que ocurren durante la ejecución de un programa. Esto incluye errores como división por cero o un archivo solicitado que no se

encuentra. El manejo de errores permite que los programas continúen ejecutándose incluso cuando ocurren errores, mejorando su confiabilidad y solidez.

Los conceptos que acabamos de enumerar son fundamentales para comprender cómo funcionan los lenguajes de programación y cómo escribir código eficiente y mantenible.

Sintaxis de los lenguajes de programación

La *sintaxis* de un lenguaje de programación se refiere a reglas para escribir declaraciones y expresiones de programas. Es importante que la sintaxis esté *bien definida* y *consistente*, para que el programador pueda escribir y comprender su código de manera efectiva. Los siguientes son los componentes básicos de la mayoría de los lenguajes de programación:

Procedimientos y funciones

Los procedimientos y funciones se utilizan para definir bloques de código reutilizables que se pueden llamar varias veces.

Variables

Las variables representan partes de la memoria y almacenan datos que pueden manipularse y pasarse entre procedimientos y funciones.

Operadores

Los operadores son palabras clave o símbolos (como `+` y `-`) que asignan valores a variables y realizan operaciones aritméticas.

Estructura de control: Generalmente, el código del programa se ejecuta en el orden en que se escribe, pero las *declaraciones condicionales* cambian el flujo de ejecución. El código que se ejecuta a continuación depende de diversas condiciones, como el contenido de la memoria, el estado del teclado, los paquetes que llegan de la red, etc. La *declaración de bucle*, una forma especial de declaración condicional, es útil para realizar las mismas operaciones en una serie de conjuntos de datos. Una *excepción*, que invoca un código especial cuando ocurre un error, es otra estructura de control.

La sintaxis y el comportamiento de estas construcciones pueden variar entre lenguajes de programación y la elección del lenguaje puede tener un gran impacto en la legibilidad y mantenibilidad del código.

Librerías

Un buen lenguaje de programación debería facilitar el desarrollo de programas y la reutilización del código existente. Muchos lenguajes de programación tienen un mecanismo para organizar procedimientos y funciones en partes que pueden reutilizarse en otros programas.

Una *librería* es una colección de procedimientos y funciones que respaldan una característica u objetivo particular, combinados en un solo archivo. La disponibilidad de muchas librerías sencillas de usar, es otro requisito importante de un buen lenguaje de programación. Por ejemplo, Python es ampliamente reconocido como un buen lenguaje para desarrollar programas relacionados con la IA porque tiene varias librerías adecuadas para el procesamiento de la IA. Con el tamaño y la complejidad de los programas, las librerías como componentes básicos prefabricados, son cada vez más importantes. Esto es especialmente cierto en el mundo del código abierto, donde los desarrolladores se sienten cómodos reutilizándo código que otros han creado. Como resultado, se ha desarrollado un ecosistema de librerías para cada lenguaje de programación, y administradores de paquetes como "composer" para PHP, "pip" para Python y "gems" para Ruby facilitan la instalación de librerías.

Las librerías también son lenguajes compilados. La combinación de varios archivos binarios y bibliotecas precompiladas para obtener un archivo único ejecutable se denomina *enlace*, y la herramienta que realiza esta operación se llama *enlazador*. Hay dos tipos de vinculación: *enlace estático*, en el que solo se incluye el código de la librería necesario en el archivo ejecutable de la aplicación final, y *enlace dinámico*, en el que todas las aplicaciones que usan esa biblioteca comparten una librería instalada en el sistema. Actualmente, los enlaces dinámicos son el enfoque preferido y se caracterizan por archivos ejecutables de aplicaciones más pequeños y menos uso de memoria en tiempo de ejecución.

Hay que tomar en cuenta que, dado que varios programas pueden utilizar las mismas librerías, las diferencias entre las versiones de una biblioteca pueden ser un problema aún mayor que las aplicaciones. Hagamos un análises por un momento y recordemos cómo mirar los números de versión. Comúnmente se utiliza el *control de versiones semántico*, que indica las versiones mediante tres números separados por puntos. Una versión típica podría ser `2.39.16`, que indica una versión principal - 2 (un número que probablemente cambie sólo una vez cada pocos años), una versión menor - 39 dentro de la versión principal (que puede actualizarse cada pocos meses a contienen importantes cambios de características) y una revisión de rápido movimiento - 16 (que puede cambiar debido a una única corrección de error). Las versiones y revisiones posteriores tienen números más altos.

Un ejemplo muy simple

Veamos un ejemplo *muy* simple de un programa de computadora en el lenguaje Python para tener una idea aproximada de algunos de los elementos mencionados.

En lenguaje natural, se supone que el programa debe hacer lo siguiente: “Solicite al usuario que ingrese un número y verifique si este número es par o impar. Finalmente, genere el resultado.”

Y aquí está el código que podemos guardar en el archivo `simpleprogram.py`:

```
num = int(input("Enter a number: "))
if (num % 2) == 0:
   print("The given number is EVEN.")
else:
   print("The given number is ODD.")
```

Incluso en estas pocas líneas de código podemos encontrar muchas de las características y elementos de sintaxis mencionados anteriormente:

1. En la línea 1 configuramos la *variable* `num` y le asignamos un *valor* con el *operador* `=`.
2. El valor asignado corresponde a la *entrada* del usuario (a través de la *función* `input()`). Además, la función `int()` garantiza que esta entrada se convierta al *tipo de dato* entero, si es posible. La expresión que se pasa a una función entre paréntesis se llama *parámetro* o *argumento*.
3. Si el usuario ingresa una cadena de texto, Python imprimiría un error como parte de su *manejo de errores*. Si se ingresa un número decimal, la función `int()` lo convierte al número base, ejemplo: `5,73` a `5`.
4. En las siguientes líneas, la estructura de control que inicia una *condición*, con las palabras clave `if` y `else`, controla lo que sucede en cada uno de los dos casos posibles (el número es par o impar).
5. Primero, el operador de módulo prueba si (`if`) al dividir el número ingresado por 2 se obtiene el valor 0 (es decir, sin resto); en este caso, el número es par. El doble `==` es el operador de comparación "es igual a", que es diferente del operador de asignación `=` en la línea 1.
6. En el otro caso (`else`), es decir, cuando la división entre 2 produce un resultado distinto de 0, el número ingresado debe ser impar.
7. En ambos casos, la función `print()` devuelve el resultado como *salida* en forma de texto.

Y así es como se ve cuando ejecutamos el programa en la línea de comandos:

```
$ python simpleprogram.py
Enter a number: 5
The given number is ODD.
```

Cuando considera cuánta lógica del lenguaje ya está involucrada en este pequeño ejemplo, se hace una idea de lo que es capaz de hacer un software complejo distribuido en miles de archivos; por ejemplo, *sistemas operativos* como Microsoft Windows, macOS o Linux, que ponen a disposición todo el hardware de una computadora y al mismo tiempo garantizan que los usuarios puedan

instalar todas las demás aplicaciones deseadas y usarlas para trabajar o divertirse.

Código de máquina, lenguaje ensamblador y ensambladores

Como se mencionó anteriormente, el hardware puede ejecutar directamente solo una serie de patrones de bits llamados código de máquina. La CPU lee un patrón de bits de la memoria en unidades de una palabra (8 a 64 bits) y ejecuta la instrucción correlacionada. Las instrucciones individuales son bastante simples, por ejemplo, “copiar el contenido de la ubicación de memoria A en la ubicación de memoria B”, “multiplicar el contenido de la ubicación de memoria C por el contenido de la ubicación de memoria D” o “leer los datos que ha llegado al dispositivo en la dirección X.” En la era de las CPU de 8 bits, algunas personas podían memorizar todos los patrones de bits utilizados en el código de máquina y escribir programas directamente. Hoy en día, el número de patrones de instrucción ha aumentado y tratar de recordar todos estos patrones no es práctico.

El código de máquina es una secuencia de patrones de bits, 0 y 1, que no es nada intuitiva para los humanos. Para hacer la programación más intuitiva, se creó el *lenguaje ensamblador*, en el que las instrucciones recibían nombres y podían especificarse mediante cadenas. En lenguaje ensamblador, las instrucciones que corresponden uno a uno al código de máquina son escritas una sola vez. Las instrucciones podrían verse así:

```
move    [B], [A]           Copiar contenido de la memoria A a la memoria B
multi   R1, [C], [D]    Multiplicar el contenido de la memoria C por el contenido de la
memoria D
input   R1, [X]         Leer los datos que han llegado al dispositivo en la dirección X
```

Una instrucción en lenguaje ensamblador corresponde a una instrucción en código de máquina, que corresponde a la instrucción exacta que el hardware puede entender y ejecutar. Las ventajas del lenguaje ensamblador sobre el lenguaje máquina incluyen:

Legibilidad y mantenibilidad mejoradas

el lenguaje ensamblador es mucho más fácil de leer y escribir que el código de máquina. Esto facilita que los programadores comprendan, depuren y mantengan su código.

Automatización del cálculo de direcciones

La programación de código de máquina también puede utilizar el concepto de variables y funciones, pero todo debe expresarse en términos de *direcciones de memoria*. El lenguaje ensamblador también asigna nombres a las direcciones de memoria, lo que facilita la expresión de la lógica de un programa.

Debido a que el lenguaje ensamblador tiene acceso a todas las funciones del hardware, se usa comúnmente en las siguientes situaciones:

Parte dependiente de la arquitectura del sistema operativo

el uso de instrucciones dedicadas que son específicas de una arquitectura de CPU, para acceder a las funciones de inicialización y seguridad, solo se puede realizar en lenguaje ensamblador.

Desarrollo de componentes del sistema de bajo nivel

El lenguaje ensamblador se utiliza para desarrollar componentes del sistema que necesitan interactuar directamente con el hardware de la computadora, como controladores de dispositivos, firmware y el sistema básico de entrada/salida (BIOS). En particular, los dispositivos de alta velocidad que requieren llevar el rendimiento del hardware al límite a menudo necesitan controladores y firmware programados en lenguaje ensamblador.

Programación de microcontroladores

El lenguaje ensamblador también se utiliza para programar microcontroladores, que son computadoras pequeñas y de baja potencia que se utilizan en una amplia gama de sistemas integrados, desde juguetes hasta control industrial. Algunos microcontroladores tienen capacidades de memoria de sólo varios cientos de bytes y normalmente se programan en lenguaje ensamblador.

El código del lenguaje ensamblador se convierte a código de máquina antes de ser ejecutado por una aplicación llamada *ensamblador*. El ensamblador es la herramienta de programación más antigua y ha aportado una serie de ventajas impensables en la programación en código máquina. Para confundir las cosas, a veces la gente se refiere al lenguaje ensamblador como ensamblador.

El código de máquina y el lenguaje ensamblador difieren de un procesador de hardware a otro. Ellos se denominan "lenguajes de bajo nivel" porque operan directamente en el hardware. Sin embargo, los conceptos de computación y entrada/salida son los mismos en todos los procesadores. Si los conceptos comunes se pueden expresar de una manera que sea más fácil de entender para los humanos, la eficiencia de la programación se puede mejorar drásticamente. Aquí es donde entran los "lenguajes de alto nivel".

Lenguajes Compilados

Los *lenguajes compilados* son lenguajes de programación que se traducen a código de máquina o a un formato intermedio llamado *código de bytes*. El código de bytes se ejecuta en la computadora destino mediante una *máquina virtual* conocida como *runtime virtual machine*. La máquina virtual traduce el código de bytes al código de máquina adecuado para cada computadora. El Bytecode permite que los programas sean independientes de la plataforma y se ejecuten en cualquier sistema con una máquina virtual compatible.

La traducción del código fuente escrito en un lenguaje de programación de alto nivel a código de máquina o código de bytes la realiza un *compilador*. Ejemplos de algunos lenguajes compilados que producen código de máquina directamente son: C y C++. Los lenguajes que producen código de bytes incluyen Java y C#.

La elección entre código de máquina y código de bytes depende de los requisitos del proyecto, como el rendimiento, el agnosticismo de la plataforma y la facilidad de desarrollo.

Lenguaje interpretado

Los *lenguajes interpretados* son lenguajes de programación que son ejecutados por un *intérprete*, en lugar de compilarse en código de máquina. El intérprete lee el código fuente y ejecuta las declaraciones contenidas en este. Un intérprete es capaz de procesar directamente el código fuente, sin transformarlo a otro formato de archivo. Así, a diferencia de un compilador, que traduce todo el programa a código de máquina antes de ejecutarlo, un intérprete lee cada línea de código y lo ejecuta inmediatamente, lo que permite al programador ver los resultados de cada línea a medida que se ejecuta.

Los lenguajes interpretados se utilizan comúnmente para *scripting*, que se refiere a programas cortos que automatizan tareas, para interfaces de línea de comandos, para control de lotes y trabajos. Los scripts escritos en lenguajes interpretados se pueden modificar y ejecutar fácilmente sin necesidad de volver a compilarlos, lo que los hace muy adecuados para tareas que requieren creación rápida de prototipos o iteraciones rápidas y flexibles. Esta conveniencia conlleva algunos inconvenientes potenciales. Por ejemplo, un programa interpretado se ejecuta más lentamente que un programa compilado (equivalente) y cualquiera que tenga el script puede acceder al código fuente.

Ejemplos de lenguajes interpretados incluyen Python, Ruby y JavaScript. Python se usa ampliamente en informática científica, análisis de datos y aprendizaje automático, mientras que Ruby se usa a menudo en desarrollo web y para crear scripts de automatización. JavaScript es un lenguaje de programación del lado del cliente integrado en los navegadores web para crear páginas web dinámicas e interactivas.

Lenguajes orientados a datos

Los *lenguajes orientados a datos* son lenguajes de programación optimizados para procesar y manipular grandes cantidades de datos. Están diseñados para manejar de manera eficiente grandes conjuntos de datos estructurados o no estructurados y proporcionar un conjunto de herramientas para trabajar con bases de datos, estructuras de datos y algoritmos para el procesamiento y análisis de datos.

Los lenguajes orientados a datos se utilizan en una variedad de aplicaciones, incluida la ciencia de datos, el análisis de big data, el aprendizaje automático y la programación de bases de datos. Son muy adecuados para tareas que implican procesar y analizar grandes cantidades de información, como limpieza y transformación de datos, visualización y modelado estadístico.

Ejemplos de lenguajes orientados a datos incluyen SQL (abreviatura de Structured Query Language), R y MATLAB. SQL es un lenguaje estándar utilizado para gestionar bases de datos relacionales y se utiliza ampliamente en los negocios y la industria. R es un lenguaje de programación y un entorno para gráficos y computación estadística. Se usa ampliamente en ciencia de datos y aprendizaje automático. MATLAB es un entorno informático numérico y un lenguaje de programación que se utiliza en una amplia gama de aplicaciones, incluido el procesamiento de señales, el procesamiento de imágenes y las finanzas computacionales.

Paradigmas de programación

Además de las características específicas de los lenguajes de programación, los *paradigmas de programación* determinan el enfoque de una solución particular. Podemos pensar en un paradigma como una estrategia básica con la que abordamos una tarea, dependiendo de los requisitos y condiciones específicas.

Un ejemplo comparable es la construcción de una casa: si los albañiles levantan las paredes ladrillo a ladrillo o si los componentes de hormigón prefabricados se ensamblan en el sitio, es una decisión fundamental que depende de las necesidades y circunstancias. ¿Qué características quieres que tenga la casa? ¿Donde está localizada? ¿Está conectada con otras casas?

De manera similar, los paradigmas marcan la dirección de la programación: si por ejemplo, un proyecto de software se divide en partes más pequeñas y separadas; y en como lo harían. Cada lenguaje de programación se adapta mejor a algún paradigma particular. Por tanto, la elección del paradigma está estrechamente relacionada con la elección del lenguaje de programación.

Los siguientes paradigmas son comunes en la programación:

Programación orientada a objetos (POO): La POO se basa en el concepto de *objetos*, que son instancias de *clases* que encapsulan datos y comportamiento. Por ejemplo, un lenguaje podría ofrecer un rectángulo como clase para ayudar al programador a mostrar un cuadro en la pantalla.

+ La programación orientada a objetos se centra en la manipulación de datos a nivel de objeto. La programación orientada a objetos facilita la escritura de código que sea mantenible, reutilizable y extensible, y se usa ampliamente en software de escritorio, videojuegos y aplicaciones web. Ejemplos de lenguajes de programación orientados a objetos incluyen Java, C# y Python.

Programación de procedimientos

La programación de procedimientos realiza tareas a través de *procedimientos*, o bloques de código que se pueden ejecutar en un orden específico. Esto facilita la escritura de código estructurado que sea fácil de seguir, pero puede generar un código menos flexible y más difícil de mantener a medida que crece de tamaño y la complejidad del proyecto. Ejemplos de lenguajes de programación procedimentales incluyen C, Pascal y Fortran.

Hoy en día se utilizan otros enfoques para el desarrollo de software, para los cuales algunos lenguajes son más adecuados que otros. Además, las interfaces de arrastrar y soltar permiten a los no programadores escribir programas, y muchos servicios en línea han comenzado recientemente a generar código a través de inteligencia artificial cuando se les dan instrucciones en lenguaje sencillo.

En conclusión, cada paradigma de programación tiene sus propias fortalezas y debilidades; la elección del paradigma a menudo depende de las necesidades del proyecto, la experiencia y preferencias del desarrollador, así como las limitaciones de la plataforma y el entorno de desarrollo. Comprender los diferentes tipos de paradigmas puede ayudarle a elegir el paradigma adecuado para sus necesidades y también puede ayudarle a escribir código mejor y más eficiente.

Ejercicios guiados

1. ¿Cuál es el propósito de las funciones?

2. ¿Cuál es la ventaja del código de bytes sobre un archivo de código de máquina?

3. ¿Cuál es la ventaja de un archivo de código de máquina sobre el código de bytes?

Ejercicios de exploración

1. ¿Cuáles son algunas de las desventajas de dividir un programa en una gran cantidad de procesos o tareas?

2. Ha encontrado varios paquetes de código abierto, ofrecidos en diferentes versiones, que brindan las funciones que necesita para su programa. ¿Cuáles son algunos criterios para elegir un paquete?

3. Además de la POO y los paradigmas de desarrollo de procedimientos, ¿qué otros enfoques de desarrollo de software existen y cuál es el lenguaje de programación que mejor soporta cada enfoque?

Resumen

En esta lección has aprendido qué es el software y cómo se desarrolla con la ayuda de lenguajes de programación. Los numerosos lenguajes de programación se diferencian no sólo en su sintaxis, sino también, por ejemplo, en la gestión de recursos de hardware o en el manejo de estructuras de datos.

Los lenguajes de programación también difieren en cómo un intérprete o compilador convierte el código fuente, legible por humanos, al código de máquina final que será procesado por la computadora.

Los paradigmas de programación determinan la estrategia de los proyectos de software y, con ello, también la elección de los lenguajes de programación adecuados, en función de las necesidades y el tamaño del proyecto respectivo.

Respuestas a ejercicios guiados

1. ¿Cuál es el propósito de las funciones?

   Las funciones encapsulan ciertas actividades comunes, como generar una cadena de texto. Al crear una función, puede permitir que su programa y otros programas realicen la función de manera conveniente y repetida sin tener que escribir su propio código.

2. ¿Cuál es la ventaja del código de bytes sobre un archivo de código de máquina?

   El archivo de código de bytes se puede ejecutar en muchas computadoras diferentes, donde una máquina virtual convierte el programa en código de máquina. Por ejemplo, JavaScript se ejecuta en muchos navegadores y en muchos tipos de computadoras.

3. ¿Cuál es la ventaja de un archivo de código de máquina sobre el código de bytes?

   El código de máquina se ejecuta lo más rápido posible. El código de bytes se ejecuta más lentamente porque la máquina virtual debe convertirlo en código de máquina mientras ejecuta el código de bytes.

Respuestas a los ejercicios de exploración

1. ¿Cuáles son algunas de las desventajas de dividir un programa en una gran cantidad de procesos o tareas?

   Cuando un programa se divide en procesos, estos deben comunicarse entre sí. Si trabajan con una gran cantidad de datos en común, los procesos podrían generar una gran sobrecarga en el intercambio de datos y en protegerlos de cambios múltiples y simultáneos (condiciones de carrera). Los procesos también incurren en gastos generales cuando se inician y finalizan. Mientras más procesos tenga, más complejo se volverá el programa y sus interacciones, por lo que los errores pueden ser más difíciles de encontrar.

   El paradigma funcional tiende a facilitar la división de programas en muchos procesos debido a la inmutabilidad. Los datos inmutables no sufren condiciones de carrera.

2. Ha encontrado varios paquetes de código abierto, ofrecidos en diferentes versiones, que brindan las funciones que necesita para su programa. ¿Cuáles son algunos criterios para elegir un paquete?

   Consulte los informes de errores y los avisos de seguridad de los paquetes, porque algunos tienen muchos errores e incluso son inseguros. A veces la última versión no es la mejor, porque es posible que se haya introducido un fallo de seguridad en la misma.

   Examine el foro donde los desarrolladores hablan sobre el paquete para ver si se mantiene activamente. Su programa probablemente estará en uso durante mucho tiempo y usted desea que el paquete también esté disponible y sea sólido con el tiempo.

   Pruebe diferentes paquetes para comprobar su rendimiento, así como su corrección.

   La mayoría de los paquetes dependen de funciones que se encuentran en otros paquetes (*dependencias*), por lo que una debilidad en una de las dependencias puede afectar su programa.

3. Además de la POO y los paradigmas de desarrollo de procedimientos, ¿qué otros enfoques de desarrollo de software existen y cuál es el lenguaje de programación que mejor soporta cada enfoque?

   Además de los paradigmas de desarrollo de procedimientos y POO, otros tipos incluyen:

   La programación funcional enfatiza el uso de funciones y conceptos matemáticos, como lambdas y cierres, para escribir código que se base en la evaluación de expresiones en lugar de la ejecución de declaraciones. La programación funcional trata las funciones como ciudadanos

de primera clase, de modo que puedan ser manipuladas por el programa, y enfatiza la *inmutabilidad*, o trabajar con variables que no pueden cambiar después de haber sido configuradas inicialmente. Esto hace que sea más fácil razonar y probar el código, así como escribir aplicaciones simultáneas y paralelas. Ejemplos de lenguajes de programación funcionales incluyen Erlang, Haskell, Lisp y Scheme.

Los lenguajes imperativos se centran en las declaraciones necesarias para controlar el flujo de las transiciones del programa desde y hacia varios estados.

Los lenguajes declarativos describen qué acciones tomar y la lógica detrás de las instrucciones. No se especifica el orden en que se llevan a cabo las instrucciones. Los compiladores pueden reordenar y optimizar estas instrucciones, llamadas a funciones y otras declaraciones, siempre que se mantenga la lógica subyacente.

La programación natural es un paradigma de programación que utiliza lenguaje natural u otras representaciones amigables para los humanos, y así describir el comportamiento deseado de un programa. La idea es hacer que la programación sea accesible para personas que quizás no tengan una formación formal en informática. Ejemplos de lenguajes de programación naturales incluyen Scratch y Alice.

051.2 Arquitectura de software

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 051.2

Peso

2

Áreas de conocimiento clave

- Comprender los conceptos de informática cliente y servidor.
- Comprender los conceptos de clientes delgados y gordos.
- Comprender los conceptos de monolitos y microservicios, así como sus principales diferencias.
- Comprender los conceptos de interfaces de programación de aplicaciones (API)
- Comprender el concepto de componentes de software y su integración o separación (servicios, módulos, API)

Lista parcial de archivos, términos y utilidades

- Clientes y servidores
- Clientes ligeros y clientes pesados
- Aplicaciones web
- Aplicaciones de una sola página
- Arquitecturas monolíticas
- Arquitecturas de microservicios
- Interfaces de programación de aplicaciones (API)
- API RESTful

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	051 Fundamentos del software
Objectivo:	051.2 Arquitectura del software
Lección:	1 de 1

Introducción

El Internet es común en nuestro mundo moderno, al igual que las aplicaciones móviles y las aplicaciones web. Una gran parte de la población mundial las utilizan para diferentes funciones, esto va desde la mensajería instantánea hasta actividades más complejas, como la compra de equipos mineros para grandes empresas.

Detrás de todas estas interfaces y servicios en línea aparentemente sencillos hay una *arquitectura* — un conjunto de piezas de software que cooperan entre sí — que a menudo damos por sentada. Hay que conocer un poco esta arquitectura para comprender cómo encajan todas las piezas de Internet y cómo el software puede aportar valor real a sus usuarios.

En esta lección, veremos algunas de las arquitecturas de software que hay detrás de las aplicaciones web; que son los sistemas de software basados en servidores (server-based software systems), y cómo se utilizan en los sistemas con los que casi todos estamos familiarizados.

Servidores y Clientes

Cuando utiliza sistemas online, es muy probable que en algún momento se haya encontrado un mensaje como el de Ejemplo cuando una respuesta esta marcha.

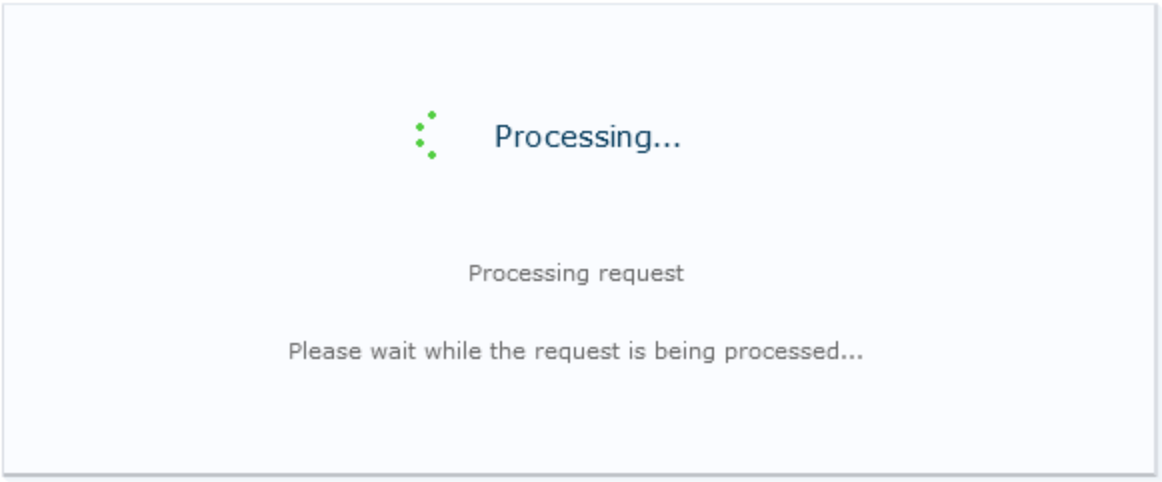

Figure 1. Ejemplo cuando una respuesta esta marcha

Retrocedamos un poco y observemos el contexto en el que está viendo este mensaje. Supongamos que está intentando obtener acceso a su cuenta bancaria a través del sitio web de su banco. Cuando intenta acceder al sitio web bancario desde su computadora portátil, utiliza un tipo de software (es decir, aplicación) llamado navegador web, como Google Chrome o Firefox. En este caso, el navegador de su computadora envía una solicitud a otra computadora que aloja el sitio web. Este tipo particular de computadora se llama *servidor*. Está especialmente diseñado para funcionar las 24 horas del día, los 7 días de la semana, y siempre atiende nuevas solicitudes provenientes de todas partes del mundo.

Entonces, un servidor es una computadora, igual a la que se usas para trabajar, jugar videojuegos y realizar tareas de programación. Sin embargo, existe una diferencia principal: un servidor normalmente utiliza todos sus recursos para la aplicación de software que se ejecuta en este. En este ejemplo, el software es una aplicación web, un programa informático que se ejecuta en el servidor.

En Ejemplo cuando una respuesta esta marcha, el servidor ha recibido una solicitud de un navegador y la aplicación que se ejecuta en el servidor está procesando la operación resultante. Esta operación podría consistir en consultar una base de datos para obtener los datos de un

usuario en el banco o comunicarse con otro servidor para verificar un descuento especial en el próximo pago del usuario.

En este ejemplo, llamamos al navegador que se ejecuta en su máquina la *aplicación cliente*, o simplemente, el *cliente*. El cliente interactúa con el servidor remoto.

La comunicación de red entre el cliente y el servidor puede tener lugar dentro de una red empresarial o a través de la red mundial que llamamos Internet. Una característica común de la interacción cliente-servidor es que un servidor puede establecer múltiples relaciones con múltiples clientes. Pensemos en el ejemplo anterior: el sitio web de un banco alojado en un servidor puede atender miles de solicitudes por minuto desde múltiples ubicaciones, cada una con un usuario diferente intentando acceder a su cuenta bancaria personal.

No todos los escenarios están estructurados como un navegador que interactúa con un servidor que realiza casi todo el procesamiento. En algunos casos, el cliente puede ser la instancia principal de procesamiento; este concepto se denomina *cliente pesado* (o *thick client*), donde el cliente almacena y procesa la mayoría de las tareas en lugar de depender de los recursos del servidor. En nuestro ejemplo bancario, por el contrario, el navegador es un *cliente ligero* (o *thin client*), que depende del servidor para calcular y devolver información a través de la red.

Un ejemplo de cliente pesado es una aplicación (de escritorio) de videojuego, donde la mayor parte del almacenamiento y procesamiento de datos se realiza localmente, utilizando la GPU, RAM, CPU y espacio en disco de la computadora para procesar la información. Una aplicación de este tipo rara vez depende de un servidor externo, eespecialmente si el juego se juega/disfruta sin conexiónA difierencia de SPA.

Ambos enfoques tienen ventajas y desventajas: para un cliente pesado, la inestabilidad de la red es un problema menor en comparación con un cliente ligero que depende de un servidor remoto, pero las actualizaciones de software pueden ser más difíciles de aplicar y el cliente pesado requiere más recursos informáticos. Para un cliente ligero, los costos más bajos podrían ser una gran ventaja. Para cualquier tipo de cliente, proporcionar datos personales a una aplicación de terceros puede ser un problema.

Aplicaciones Web

Una *aplicación web* es un software que se ejecuta en un servidor; procesa las interacciones del usuario y es contactado por clientes pesados o ligeros a través de una red informática. No todos los sitios web se consideran aplicaciones web: las páginas web estáticas simples sin interactividad no se consideran aplicaciones web porque el servidor no ejecuta una aplicación para procesar las acciones solicitadas por el cliente.

Muchas aplicaciones web se pueden dividir en dos grupos: la *aplicación de una sola página* (SPA - single page application) y la *aplicación de varias páginas* (MPA - multi page application). Una SPA tiene una sola página web, donde se produce todo el intercambio y carga de datos sin necesidad de redirigir al usuario a otra página web dentro de la aplicación. A diferencia de SPA, una MPA tiene múltiples páginas web. Un cambio de datos podría actualizar la misma página web que originó la acción o redirigir al usuario a otra página web.

Considere el ejemplo anterior, donde un usuario desea verificar las transacciones de su cuenta más recientes en el sitio web de su banco. Imagine que se produce una transacción después de cargar la página web. Si la aplicación web del banco es un SPA, la nueva transacción se mostrará automáticamente en la misma página web, sin redirigir al usuario a una nueva página. Si el usuario consulta sus préstamos, la nueva información también se muestra en la misma página, evitando la necesidad de redirección. Modificar la página sin redirigir al usuario facilita la navegación. Para un MPA, cuando el usuario solicita la página web de préstamo, el servidor debe redirigir al usuario a una nueva ubicación, es decir, a otra página web.

Un famoso ejemplo de aplicación web SPA, es Google Mail (Gmail). No redirige al usuario a una página completamente nueva cuando, el usuario desea mostrar la carpeta de spam, ya que la aplicación simplemente actualiza la parte específica de la pantalla que muestra todos los mensajes de spam y permanece en la misma página web.

Por otro lado, una aplicación MPA es Amazon.com, el gigante del comercio electrónico, donde cada artículo se ubica en una página web distinta.

Una ventaja de una MPA sobre una SPA es que los análisis web son mucho más fáciles de recopilar y medir. Esto es crucial para ayudar a los desarrolladores a optimizar los resultados de búsqueda en Internet. Normalmente, una aplicación web se divide en dos partes: *frontend* y *backend*. El fronted (o interfaz) es la capa de vista, donde el usuario interactúa con un navegador utilizando los elementos de la página haciendo clic, seleccionando o escribiendo. Aquí es donde los datos del servidor se reciben, se formatean y se muestran al usuario a través del navegador.

El backend es generalmente la parte más grande de una aplicación web. Comprende la lógica empresarial, los controladores de comunicación, la mayor parte del procesamiento de datos y el almacenamiento de estos. El almacenamiento de datos se realiza a través de un sistema de gestión de bases de datos independiente conectado al backend.

El frontend y el backend se comunican entre sí. Las solicitudes de datos son reenviadas por el frontend al backend, y el frontend recibe, formatea y muestra los datos devueltos por el backend al usuario.

En nuestro ejemplo anterior de obtener la última transacción en la cuenta bancaria de un usuario, la acción es interpretada por el frontend de la aplicación, que se ejecuta en el navegador del

escritorio del usuario. Luego, esta solicitud se envía al backend de la aplicación, que valida si el usuario tiene permiso para realizar la acción, recupera los datos y devuelve la lista de transacciones al frontend cargado en el navegador. Luego, el navegador formatea y muestra los datos al usuario.

Interfaz de Programación de Aplicaciones (API)

Ningún software es útil sin comunicarse con componentes internos y externos. Entonces, ¿cómo puede comunicarse el cliente con la aplicación web? ¿Cómo puede el frontend enviar datos al backend?

Las aplicaciones de software se comunican entre sí a través de una *interfaz de programación de aplicaciones* (API - Application Programming Interface), que se ejecuta a través de *protocolos* básicos de comunicación de Internet. Los protocolos son estándares y reglas desarrollados para garantizar que dos o más aplicaciones intercambien comandos y datos.

El principal beneficio de una API es desacoplar diferentes partes de la aplicación y al mismo tiempo permitirles cooperar en el procesamiento de datos. Las API también centralizan el flujo de datos en canales predefinidos, actuando como una puerta de enlace que garantiza que todos utilicen el mismo camino para ir y venir. En las aplicaciones web, las API son vitales para la funcionalidad de la aplicación, ya que permiten la interacción del usuario, la entrega de información procesada, solicitudes de almacenamiento de datos y muchas otras tareas. El cliente puede utilizar una API para solicitar acciones que se ejecutarán en el servidor, por ejemplo.

Volvamos al ejemplo de la aplicación web bancaria. Para iniciar sesión en una cuenta a través de una aplicación web, el usuario generalmente escribe datos como nombre de usuario y contraseña en ciertos campos de texto y luego hace clic en el botón "Iniciar sesión". El navegador toma esta información y llama a una API de backend. La aplicación web que se ejecuta en el servidor remoto recibe los datos del usuario, los valida, verifica el derecho del usuario a obtener acceso y finalmente envía una respuesta al navegador. Para que el usuario se comunique con el servidor, es obligatorio que tanto el cliente como el servidor envíen datos de un lado a otro. Eso es lo que permiten las API.

Tenga en cuenta que la aplicación web del banco no expone otra información confidencial; muestra al usuario solo los campos que están permitidos y son necesarios para una interacción deseada. El resto está oculto al usuario.

La comunicación entre API puede basarse en diseños y protocolos muy diferentes. El *protocolo de transferencia de hipertexto* (HTTP - Hypertext Transfer Protocol) es el protocolo más utilizado en aplicaciones web. *Hipertexto* es texto con enlaces a otros textos, concepto que subyace a los enlaces en las páginas web HTML. Los hipervínculos constituyen así la base para construir

páginas web y mostrarlas en los navegadores.

HTTP fue diseñado para aplicaciones cliente-servidor, donde los recursos se solicitan desde un servidor y luego se devuelven al cliente a través de la red utilizando una estructura predefinida especificada por el protocolo HTTP.

Para una aplicación web estructurada, los ingenieros de software diseñan la aplicación con partes o módulos separados. Esta separación de responsabilidades permite tareas y responsabilidades claramente definidas, resultando así un desarrollo más rápido y un mejor mantenimiento. Tomemos, como ejemplo una aplicación con dos módulos internos: uno que implementa la lógica empresarial y el otro que depende de una integración de terceros. Este tercero es una empresa externa que proporciona su API para algún propósito específico, ejemplo: pronóstico del tiempo. Si el servidor meteorológico no funciona, es imposible obtener detalles meteorológicos, y si estos datos son cruciales para el resultado final, el usuario puede experimentar algunos dolores de cabeza temporales si no hay una fuente alternativa para los datos.

Ahora imagine que se reemplaza este proveedor externo y el nuevo tiene una forma diferente de manejar la misma API. La separación de módulos significa que los desarrolladores tendrán que actualizar solo ese módulo. La lógica de negocios en el otro módulo no necesita ser tocada en absoluto, o al menos requiere actualizaciones mínimas.

La necesidad de estructuras de procesos claras también influye en el diseño de las API para facilitar su uso. El concepto de *transferencia de estado representacional* (REST - Representational state transfer) es un estilo de arquitectura con un conjunto de pautas para diseñar e implementar el acceso a los datos en una aplicación.

Hay seis principios REST. Para simplificar, vamos a explicar tres que son más relevantes para esta lección:

Desacoplamiento cliente-servidor

El cliente debe conocer únicamente el URI del recurso a través del cual se produce la comunicación con el servidor. Este principio permite una mayor flexibilidad. Por ejemplo, cuando se refactoriza el lado backend de la aplicación, o hay un cambio de arquitectura importante en una base de datos backend, no es necesario actualizar el frontend en conjunto. Simplemente continúa enviando las mismas solicitudes HTTP al backend.

Sin estado

Cada nueva solicitud es independiente de las anteriores. No es casualidad que el protocolo HTTP sea ampliamente utilizado para aplicaciones que siguen principios REST, ya que HTTP no tiene conocimiento de solicitudes HTTP anteriores. Para cada nueva solicitud se deberá enviar toda la información necesaria para poder procesar correctamente la solicitud. Por ejemplo, una

aplicación web que implementa este principio no sabe si el cliente ha iniciado sesión (autenticado). Entonces, para cada solicitud HTTP, el cliente debe enviar un token de autenticación, y el servidor podrá usar este token para verificar si la solicitud debe bloquearse o procesarse.

Una de las principales ventajas de este principio es que es más fácil de escalar, porque el servidor puede procesar millones de solicitudes sin verificar los detalles del usuario.

Arquitectura en capas

La aplicación se compone de varias capas y cada capa puede tener su propia lógica y propósito, como seguridad o adquisición de datos. Es posible que el cliente nunca sepa cuántas capas existen o si se están comunicando directamente con una capa específica dentro de la aplicación.

Las API que siguen los principios REST se denominan *RESTful* y, en la web moderna, muchas aplicaciones web siguen el diseño REST. Aunque una API RESTful no necesita implementar estos principios utilizando el protocolo HTTP, se usa casi universalmente en el modelo REST dada su solidez, simplicidad y universalización en el entorno web.

Tipos de arquitectura

Existen docenas de estilos y estándares de arquitecturas que intentan organizar las estructuras de las aplicaciones web. Como casi todo en el mundo de la informática, no hay un "ganador", sólo un conjunto de pros y contras para cada modelo. Un paradigma importante es la llamada *arquitectura de microservicios*, que se creó como alternativa a la antigua *arquitectura monolítica*.

La arquitectura de microservicios es un modelo de software compuesto por múltiples *servicios* interdependientes que juntos forman la aplicación final. Esta arquitectura tiene como objetivo descentralizar el código base en múltiples capas de software que se dividen en múltiples aplicaciones más pequeñas para un mejor mantenimiento (Arquitectura de microservicio).

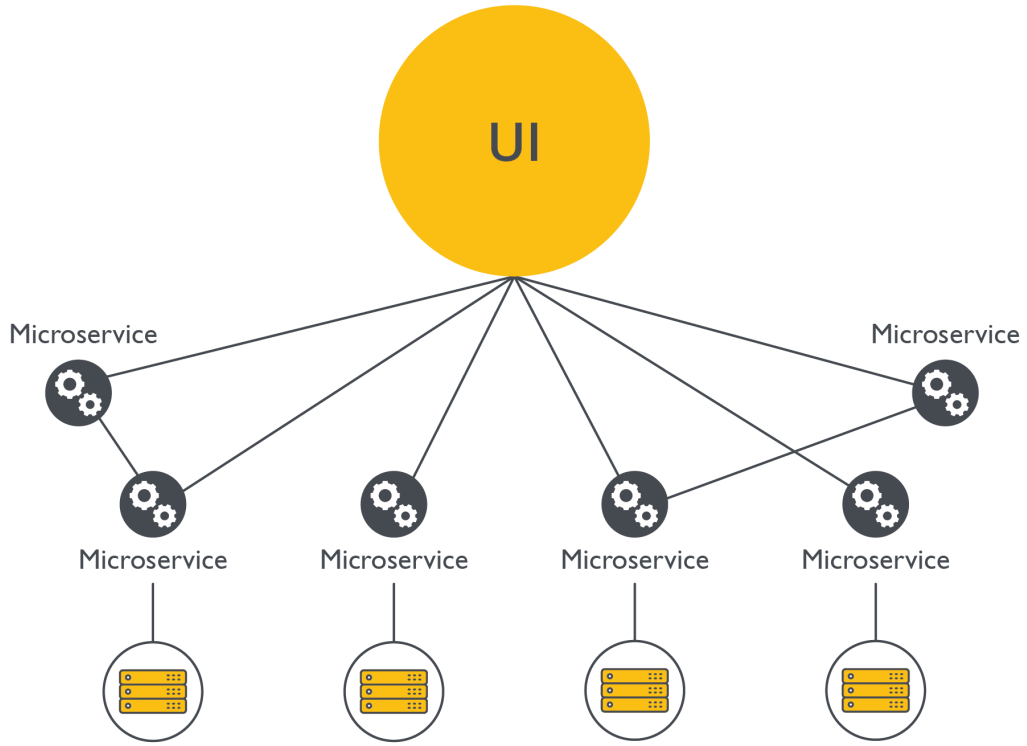

Figure 2. Arquitectura de microservicio

Por el contrario, una arquitectura monolítica contiene todos los servicios y recursos de la aplicación en una sola aplicación (Arquitectura monolítica).

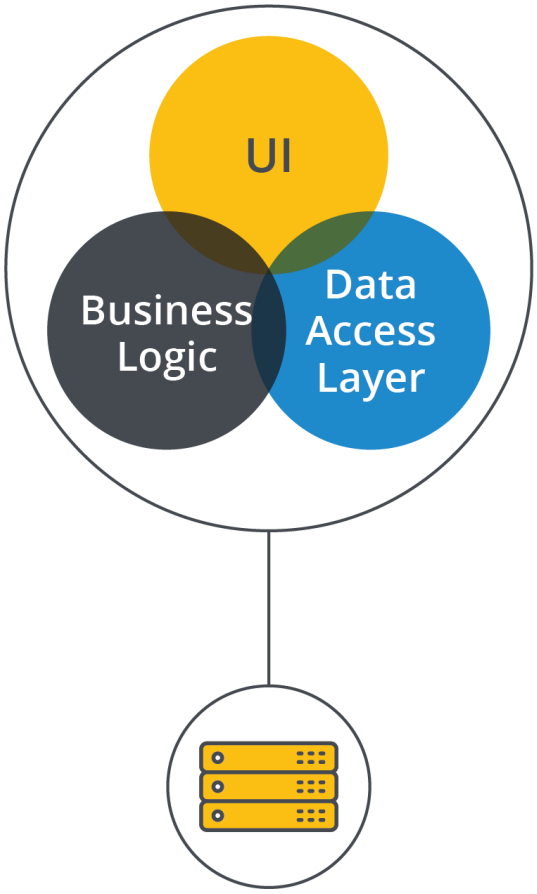

Figure 3. Arquitectura monolítica

Las imágenes muestran que el modelo de microservicio está descentralizado y la aplicación depende de múltiples servicios, cada uno con su propia base de datos, código base e incluso recursos de servidor. Como su nombre lo indica, cada microservicio debe ser más pequeño que su contraparte monolítica, que asume la responsabilidad de todos los servicios.

La aplicación monolítica encapsula todos sus recursos en una sola unidad. Toda la lógica empresarial, los datos y el código base están centralizados en un bloque enorme, por lo que se llama “monolito”.

Los usuarios apenas notan si una aplicación web se está ejecutando como un modelo monolítico o de microservicio; la elección debe ser transparente. Nuestra aplicación bancaria, por ejemplo, podría ser un monolito donde toda la lógica empresarial relativa a pagos, transacciones, préstamos, etc. esté ubicada en el mismo código base ejecutándose en uno o más servidores. Por otro lado, si la aplicación bancaria utiliza un estilo de microservicio, probablemente tenga un microservicio dedicado a procesar pagos y otro microservicio solo para emitir préstamos. Este

último microservicio llama a otro microservicio más para analizar la probabilidad de que el solicitante incumpla el pago. La aplicación podría tener miles de servicios más pequeños.

El enfoque monolítico requiere una mayor sobrecarga de mantenimiento cuando la aplicación crece, especialmente con varios equipos codificando en la misma base de código. Dados los recursos de software centralizados, es muy probable que un equipo cambie algo que rompa la parte de la aplicación del otro equipo. Esto podría ser un verdadero dolor de cabeza para los equipos más grandes, especialmente cuando hay una gran cantidad de estos.

Los microservicios son mucho más flexibles en ese sentido, porque cada servicio es gestionado por un solo equipo. Por supuesto, un equipo puede gestionar más de un servicio. Los cambios de código se realizan fácilmente y los recursos en competencia no son un problema real. Dado que cada servicio está interconectado, cualquier punto de falla podría tener un impacto negativo en toda la aplicación. Además, dado que existen múltiples instancias de bases de datos, servidores y API externas que se comunican entre sí, la resiliencia de toda la aplicación es tan buena como la de su microservicio más débil.

Una ventaja del enfoque monolítico es tener una fuente de datos centralizada, lo que facilita evitar la duplicación de datos. El enfoque también reduce el consumo de recursos de la nube, porque un servidor más grande necesita menos recursos informáticos que varios servidores descentralizados. Una aplicación de microservicio de aproximadamente el mismo tamaño supone una carga mayor para la nube.

Ejercicios guiados

1. ¿Cuáles son las principales diferencias entre un cliente pesado y uno ligero?

2. ¿Es correcto suponer que cada sitio web es una aplicación web?

3. ¿Qué es el modelo REST?

4. ¿Cuál es el modelo preferido para desarrollar aplicaciones web grandes y modernas con múltiples equipos de desarrollo? ¿Pór qué?

5. ¿Cuál es el protocolo más utilizado para intercambiar datos entre aplicaciones web?

6. Nombre dos desventajas de las aplicaciones de varias páginas en comparación con las aplicaciones de una sola página.

7. Describa una ventaja de un sistema monolítico sobre un sistema de microservicio y una ventaja que tiene el sistema de microservicio en comparación con uno monolítico.

Ejercicios de exploración

1. En 2021, el rover Perseverance de la NASA aterrizó en Marte, y uno de sus objetivos es determinar si alguna vez existió vida en Marte. Aunque el Rover podría controlarse desde la Tierra, también puede controlarse a sí mismo en la mayoría de situaciones. ¿Por qué es una buena idea proyectar un rover como un cliente pesado?

2. Considere un automóvil personal moderno, autónomo y que se conecta a un servidor externo para intercambiar datos. ¿Debería ser un cliente pesado o ligero?

Resumen

En está lección se explicaron los conceptos básicos de la arquitectura de software para aplicaciones web. Se detalló cómo se estructuran y organizan comúnmente, y las principales diferencias entre los modelos monolíticos y de microservicios. Cubrimos los conceptos de servidores y clientes, y los conceptos básicos de la comunicación de aplicaciones web entre clientes y otros programas de software.

Respuestas a ejercicios guiados

1. ¿Cuáles son las principales diferencias entre un cliente pesado y uno ligero?

   Un cliente pesado no requiere una conexión constante a un servidor remoto que devuelva información crítica al cliente en ejecución. El cliente ligero depende en gran medida de la información procesada por una fuente externa. Otra diferencia es que un cliente pesado es responsable de la mayor parte del procesamiento de datos, por lo que requiere más recursos informáticos que su contraparte ligera.

2. ¿Es correcto suponer que cada sitio web es una aplicación web?

   No. Hay sitios web que no son aplicaciones de software. Una aplicación web interactúa con el usuario, quien puede ingresar datos y utilizar funcionalidades web en tiempo real. Los sitios web simples, como un anuncio de un evento social, que funciona como un banner web, no son aplicaciones web. Estos sitios web no interactivos son más fáciles de mantener y requieren pequeños recursos informáticos para alojar y entregar las páginas web. Una aplicación web requiere muchos más recursos informáticos, servidores más robustos y funcionalidades que manejen a los usuarios, como acceso restringido y almacenamiento permanente de datos.

3. ¿Qué es el modelo REST?

   El modelo REST es un modelo de arquitectura de software que proporciona a las aplicaciones una guía de desarrollo para una mejor usabilidad, claridad y mantenibilidad. Uno de los principios descritos en el conjunto de directrices REST es la *arquitectura en capas*, utilizada principalmente para la cohesión y para reducir la dependencia de los componentes internos de las distintas API.

4. ¿Cuál es el modelo preferido para desarrollar aplicaciones web grandes y modernas con múltiples equipos de desarrollo? ¿Pór qué?

   El modelo de software de microservicios proporciona un marco flexible en el que los equipos colaboran en la misma aplicación de software, lo que proporciona una simultaneidad más sencilla para dos o más equipos que mantienen una aplicación web de gran tamaño. Debido a que el marco está descentralizado, cada equipo puede actualizar un dominio empresarial específico sin tener que actualizar otros componentes.

5. ¿Cuál es el protocolo más utilizado para intercambiar datos entre aplicaciones web?

   HTTP es el protocolo más utilizado para intercambiar datos y comandos entre servidores y clientes.

6. Nombre dos desventajas de las aplicaciones de varias páginas en comparación con las aplicaciones de una sola página.

   Una aplicación de varias páginas recarga todos los elementos de la página web cuando el usuario realiza algunas acciones, en lugar de actualizar solo los elementos modificados. El rendimiento se ve afectado por este diseño. Otra desventaja de un MPA es una interactividad del usuario más complicada, donde cada carga de página crea una pérdida en la facilidad de uso. Por el contrario, el efecto visual podría ser continuo en una SPA.

7. Describa una ventaja de un sistema monolítico sobre un sistema de microservicio y una ventaja que tiene el sistema de microservicio en comparación con uno monolítico.

   Un sistema monolítico puede facilitar la administración de datos porque la información se encuentra en una gran base de datos, en lugar de estar dispersa en varias bases de datos. Una aplicación de microservicios, por otro lado, puede mejorar el desarrollo y mantenimiento del código; Varios equipos pueden trabajar en diferentes lógicas de negocio sin bloquear el progreso de otros equipos.

Respuestas a los ejercicios de exploración

1. En 2021, el rover Perseverance de la NASA aterrizó en Marte, y uno de sus objetivos es determinar si alguna vez existió vida en Marte. Aunque el Rover podría controlarse desde la Tierra, también puede controlarse a sí mismo en la mayoría de situaciones. ¿Por qué es una buena idea proyectar un rover como un cliente pesado?

   El tiempo que tarda una señal de comunicación en enviarse desde la Tierra y recibirse en Marte, puede variar dependiendo de las posiciones de esos planetas, pero puede tardar hasta veinte minutos. Por lo tanto, el mando y control de un vehículo distante en movimiento es imposible, especialmente si se tienen en cuenta situaciones inesperadas. Idealmente, el rover debería controlarse a sí mismo en la mayoría de situaciones. Esto se logra mediante entrenamiento de inteligencia artificial (IA) (Machine Learning), de modo que el rover se vuelve más independiente de los comandos manuales. Para hacerlo posible y depender menos de señales distantes, se proyectó que el rover tuviera sus propios recursos y la mayoría de los procesos informáticos se ejecutaran localmente, coincidiendo con la definición de un cliente pesado.

2. Considere un automóvil personal moderno, autónomo y que se conecta a un servidor externo para intercambiar datos. ¿Debería ser un cliente pesado o ligero?

   Un vehículo autónomo podría delegar el procesamiento pesado de datos a un servidor externo y confiable, pero esto sería susceptible a períodos de desconexión cuando se requiere procesamiento de datos críticos. Por lo tanto, es imperativo que el vehículo autónomo procese la mayoría de las tareas, y esto requiere que sea un cliente pesado con redundancia múltiple.

051.3 Computación local y en la nube

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 051.3

Peso

1

Áreas de conocimiento clave

- Comprender los conceptos de computación local y en la nube.
- Comprender los modelos comunes de operación en la nube
- Comprender los tipos comunes de servicios en la nube
- Comprender los principales beneficios y riesgos de la computación en la nube y la infraestructura de TI local

Lista parcial de archivos, términos y utilidades

- Computación en la nube
- Infraestructura de TI local
- Centro de datos
- Nube pública, privada e híbrida
- Infraestructura como servicio (IaaS), Plataforma como servicio (PaaS), Software como servicio (SaaS)
- Modelos de costos
- Seguridad
- Propiedad de los datos
- Servicio de disponibilidad

Lección 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	051 Software Fundamentals
Objective:	051.3 Computación local y en la nube
Lesson:	1 of 1

Introducción

Hoy en día, todo el mundo habla de la nube. Las empresas están evaluando servicios con una variedad de proveedores, incluidas grandes empresas como Amazon y Microsoft, para ver que servicios ofrecen en la nube.

Pero la idea de la computación en la nube se remonta sólo a finales de la década del 2000. Y el término es vago, tan vago que muchas personas, incluida la Free Software Foundation, desaconsejan su uso. Sin embargo, la idea de la computación en la nube incorpora algunos conceptos útiles y es una tendencia crucial en la informática moderna. Esta lección analiza qué es la nube (en un nivel alto) y cómo funciona tanto técnica como financieramente. Además, analizaremos los beneficios y riesgos de la nube. Descubriremos, por qué este tema está relacionado con el código abierto.

Computación local y en la nube

Si trabaja o estudia en una organización que tiene más de un par de sistemas informáticos, es casi seguro que hay un *centro de datos*: una habitación separada para albergar los servidores de la organización, generalmente cerrada con llave y con aire acondicionado. Un centro de datos *on-*

premises (o *on-premise*) es simplemente un recurso informático físico que las organizaciones tienen para su propio uso.

Existen varias alternativas a la ejecución de un centro de datos local. Durante décadas, antes de la tendencia que llamamos “computación en la nube”, las empresas establecían centros de datos que albergaban computadoras en nombre de los clientes. Por lo tanto, puede otorgarle licencias a cinco servidores de la empresa y darle varias especificaciones de CPU, memoria y almacenamiento, y luego cargar su software en los servidores. Este modelo de negocio es el *alojamiento remoto*. El alojamiento remoto ofrece muchas ventajas. La organización puede subcontratar eficazmente la experiencia administrativa de la compra y configuración de servidores al negocio de alojamiento remoto, lo que podría generar compras al por mayor. En otras palabras, evita la responsabilidad de tener una infraestructura de TI local. El negocio de hosting remoto garantiza la seguridad física, liberando además al cliente de esa preocupación. Finalmente, configurar un nuevo servidor en una empresa de alojamiento remoto es mucho más rápido que comprar, enviar y configurar el servidor local. Una organización también puede mantener un centro de datos local y al mismo tiempo otorgar licencias para más servidores en el entorno remoto para recuperarse de desastres o proporcionar potencia informática adicional durante períodos de uso intensivo.

Tenga en cuenta que la informática remota supone la presencia de una red rápida y confiable. Exploraremos este problema junto con otros beneficios y debilidades en otra sección.

Todas las ventajas del alojamiento remoto se aplican también a la computación en la nube. La computación en la nube es técnicamente diferente porque no hay ninguna computadora física en particular dedicada a su organización. En cambio, el proveedor de la nube ejecuta múltiples sistemas para múltiples clientes en cada sistema físico, utilizando una capa adicional de software conocida como *máquinas virtuales*.

Por lo tanto, un servicio en la nube ejecuta un centro de datos como cualquier otra organización, pero sirve a otras organizaciones en lugar de (o además de) a sí mismo. El centro de datos almacena miles de computadoras físicas. En cada computadora física ejecuta un sistema operativo (generalmente llamado *hipervisor*) que admite múltiples máquinas virtuales. Cada máquina virtual se puede generar y eliminar rápidamente. Cada máquina virtual admite un sistema operativo ejecutado por un cliente ([Computación en la nube]).

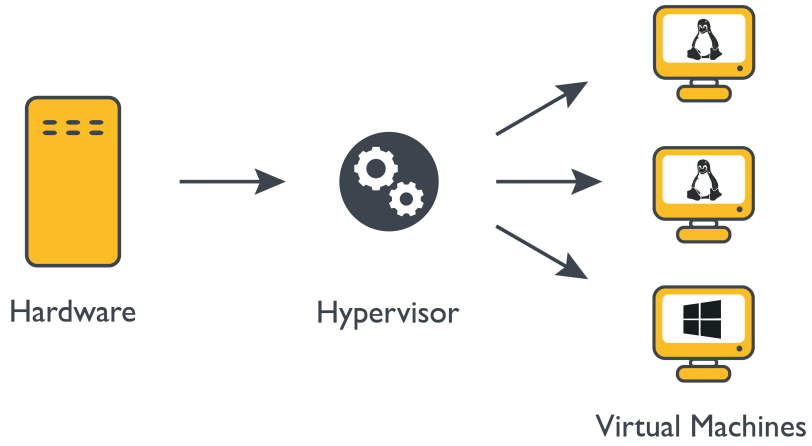

Figure 4. Cloud computing

¿Dónde entra el software libre y el código abierto? Cuando una empresa utiliza una gran cantidad de computadoras e implementa sistemas operativos, es importante no atascarse con las licencias. Aunque existen modelos de licencia para sistemas operativos propietarios en la nube, es más complicado que simplemente ejecutar una máquina virtual y un sistema operativo de código abierto. El código abierto también suele ser gratuito.

El comienzo de la computación en la nube generalmente se remonta al lanzamiento de Amazon Web Services (AWS) por parte de Amazon.com en 2006. En la actualidad existen docenas de empresas de nube, incluidas ofertas de proveedores tan grandes como Microsoft, Google, Alibaba e IBM. AWS sigue siendo la oferta más grande. Los proveedores compiten ferozmente en el desarrollo de nuevas funciones y servicios, porque todos son fuertes en costo y confiabilidad.

Las ventajas de la computación en la nube se suman a las de la computación remota. Los costos son menores porque un sistema físico puede ejecutar muchos servidores para muchos clientes y puede mantenerse ocupado constantemente. Un cliente que necesita más potencia informática rápidamente para un aumento en el uso puede generar nuevos sistemas en segundos. Los sistemas se pueden gestionar automáticamente a través de una interfaz de programación de aplicaciones (API). Nuevamente, analizaremos más de cerca los beneficios y riesgos más adelante.

Modelos comunes de operación en la nube

Antes de detallar modelos de operación, cabe señalar que muchas empresas han adoptado modelos de nube que cambian totalmente su forma de programar y ofrecer servicios. En lugar de actualizar cada aplicación una o dos veces al año, las empresas permiten actualizaciones rápidas.

Pueden hacer esto porque la nube les permite apagar máquinas virtuales e iniciar otras nuevas casi instantáneamente con la nueva versión de la aplicación. La organización también puede ampliarse y reducirse rápidamente, por lo que les gusta dividir las aplicaciones en muchas partes modulares, a veces denominadas *microservicios*.

Pero en esta sección nos centraremos en las operaciones regulares en los modelos de nube. El modelo de costos de la nube es muy diferente de los costos locales. Los centros de datos locales requieren la compra única de un servidor, junto con costos rutinarios de energía, aire acondicionado y administración. Estos desaparecen cuando obtienes licencias de sistemas de un proveedor de nube, pero se cobra por lo que usa. Los proveedores de la nube dividen el uso de su servidores en períodos de tiempo y cobran por cada período. También cobran por la cantidad de datos que almacena en sus sistemas.

Hasta ahora, hemos estado hablando de proveedores que ofrecen potencia informática a los clientes; esto se llama *nube pública*, pero también puede haber una *nube privada*. Algunas organizaciones grandes administran sus propios centros de datos locales como una nube. Proporcionan servicios sólo a sus propios departamentos o subdivisiones, pero tratan a cada uno de sus departamentos como un cliente de un proveedor de nube. El centro de datos realiza un seguimiento de cuánto tiempo de cómputo, datos, etc. Utiliza cada departamento y lo cobra por ese uso.

Muchas organizaciones utilizan múltiples servicios en la nube por diversos motivos, como protegerse contra fallas de los proveedores, mantener los datos en una determinada región geográfica o aprovechar las funciones especiales que ofrece un proveedor en particular. Además, es común mantener tanto un centro de datos local como servidores en la nube, una práctica llamada *computación híbrida*.

Un cliente que se registra en un servicio en la nube puede elegir en qué regiones geográficas ejecutar. Por ejemplo, Amazon actualmente ofrece regiones en el oeste y el este de EE. UU., varias regiones de Europa y más regiones en todas partes del mundo. Normalmente, elegirías la región más cercana a ti. Pero muchas organizaciones quieren operar en múltiples regiones porque tienen un alcance internacional. A veces, las organizaciones necesitan guardar datos en un lugar particular para cumplir con el Reglamento General de Protección de Datos de Europa (GDPR) o la Ley de Protección de Información Personal de China (PIPL).

Cada región normalmente se subdivide en *zonas* o *zonas de disponibilidad*. Se recomienda ejecutar en varias zonas en caso de que un desastre provoque la caída de una zona. Aunque la nube se caracteriza por compartir servidores físicos entre múltiples clientes, algunos proveedores pueden dedicar un solo servidor a un cliente en particular preocupado por la seguridad. Como ninguna otra organización utiliza la computadora, el cliente se siente un poco más seguro al ejecutar sus servicios confidenciales y cargar sus datos en la nube. Esta opción acerca la computación en la

nube al antiguo alojamiento remoto.

Tipos comunes de servicios en la nube

En diferentes niveles, la computación en la nube tiene un aspecto muy diferente y está dirigida a diferentes tipos de usuarios. *Infraestructura como servicio* (IaaS) es la categoría con la que suelen tratar los administradores de sistemas. IaaS proporciona solo hardware y software que admiten máquinas virtuales. Depende de los administradores del sistema del cliente, cargar un sistema operativo y las aplicaciones deseadas en la máquina virtual. El administrador del sistema maneja casi todo de la misma manera que en un centro de datos local. La plataforma como servicio (PaaS) es una invención más reciente, utilizada principalmente por programadores. Aquí, el programador no se preocupa por el sistema operativo y no tiene que cargar las bibliotecas que utiliza el programa. Todo eso lo proporciona el proveedor de la nube. El programador simplemente carga funciones que se ejecutan en la plataforma. Un concepto relacionado es el de la informática *serverless*.

Software como servicio (SaaS) es una aplicación que se ejecuta en un sistema en la nube. Cada vez que inicia sesión en un sitio de redes sociales, solicita un artículo en una tienda en línea, visita una página web para ingresar sus horas en un sistema de seguimiento de empleo o ingresa un formulario en un sitio gubernamental, está utilizando SaaS. La mayor parte de la aplicación se ejecuta en el sistema remoto y la única parte de la aplicación que se ejecuta en su computadora es la página web que muestra su navegador.

Base de datos como servicio (DaaS) a menudo se agrega a las categorías anteriores. Un servicio de base de datos, como el S3 de Amazon, fue en realidad la primera oferta en la nube. Una oferta DaaS puede ser simplemente una instancia de un servidor de base de datos popular como MySQL, Oracle o MongoDB que se ejecuta en la nube. Los grandes proveedores de nube también ofrecen bases de datos patentadas que se ejecutan únicamente en sus ofertas de nube. En cualquier caso, lees y escribes la base de datos como si la tuvieras en tus sistemas.

Algunas empresas ofrecen otras variaciones de esas categorías básicas, como Security as a Service.

Principales beneficios y riesgos de la computación en la nube y la infraestructura de TI local

Antes de examinar la computación en la nube en detalle, intentemos una analogía. Administrar un centro de datos local es como comprar una casa. Si el sótano se inunda o la caldera deja de funcionar, es necesario buscar a alguien que lo arregle. Por el contrario, el alojamiento remoto y la computación en la nube son como alquilar un apartamento: el propietario es responsable de arreglar la caldera. Además, en la nube puedes agregar y eliminar rápidamente cargas de datos,

del mismo modo que puedes cambiar de apartamento más rápidamente de lo que puedes cambiar de casa de tu propiedad. En un apartamento, el propietario podría incluso proporcionar electrodomésticos y muebles. En nuestra analogía, esto es similar a los numerosos servicios que ofrecen los proveedores de la nube, como bases de datos y análisis.

Ahora podemos analizar los beneficios y riesgos de utilizar la nube, en lugar de su propio centro de datos.

La *flexibilidad* es probablemente la razón más convincente para migrar a la nube. Si es un minorista que necesita ejecutar más servidores cerca de Navidad, o un contador que realiza los cálculos de impuestos que realiza la mayor parte de su negocio en la temporada de impuestos, preferirá la nube para activar nuevas máquinas virtuales en cualquier momento y luego eliminarlos Los *costos* pueden ser menores en la nube por varias razones. Está compartiendo un servidor físico con muchas otras aplicaciones, por lo que las computadoras se utilizan de manera más eficiente. Como los proveedores de nube son grandes, pueden lograr economías de escala en compras, administración, refrigeración y otros requisitos de infraestructura. Finalmente, los clientes quedan liberados de muchas tareas administrativas, aunque la administración del sistema no va a desaparecer en absoluto. Los clientes aún necesitan administradores de sistemas para crear y cargar su software (conocido como *instancias* en la nube), para autorizar a los usuarios y otras tareas relacionadas con las operaciones comerciales. Los administradores del sistema deben aprender la API del proveedor y las reglas para usar el servicio, un costo de capacitación que debe tener en cuenta en sus planes.

Por otro lado, hay que tener cuidado con el uso que se hace de la nube. Puede ser difícil realizar un seguimiento de la cantidad de potencia de la computadora que estás usando cuando puedes hacer funcionar los servidores rápidamente, especialmente si automatizas tu escalado. Es posible que al final del período te encuentres con una factura desagradablemente elevada.

¿Es la nube más eficiente en cuanto a emisiones de carbono que hacer funcionar nuestras propias computadoras? Las investigaciones encuentran que los proveedores de nube pueden ejecutar sus sistemas de manera mucho más eficiente que usted o yo. Pero tenemos que comunicarnos con esos sistemas a través de una red, lo que requiere mucha electricidad para alimentar todos los equipos de red. Lamentablemente, la nube aumenta nuestra huella de carbono.

La *disponibilidad del servicio* a veces es mejor con la nube. Ciertamente, si depende de su propio centro de datos local, es vulnerable a todo tipo de problemas, desde desastres naturales hasta saboteadores internos maliciosos. Pero los centros de datos en la nube también colapsan. Por lo tanto, debes aprovechar las diferentes zonas de disponibilidad y distribuir tu riesgo. Existen herramientas que le permiten cambiar sus servicios de una zona fallida a una que funciona.

Si utiliza los servicios ofrecidos por el proveedor, como una base de datos en la nube, es

vulnerable a errores en ese servicio. Por supuesto, también puedes sufrir errores en el software que tienes en tu sistema.

Un riesgo mayor al utilizar los servicios de los proveedores es el bloqueo. Por lo general, puede encontrar una herramienta de conversión automatizada para sacar sus datos del sistema del proveedor a uno nuevo, pero es posible que la herramienta no haga un trabajo completo.

La *seguridad* puede ser mejor en la nube porque el personal del proveedor probablemente sea más experto que el personal de seguridad. Por otro lado, los proveedores de nube son grandes y bien conocidos, y constituyen objetivos obvios para los ataques. Además, agregar una pieza adicional de software (el hipervisor que controla las máquinas virtuales) introduce un nuevo peligro potencial. Los investigadores han encontrado vulnerabilidades en los hipervisores.

Aunque el cliente sigue siendo el propietario legal de sus datos, almacenarlos en la nube, en teoría, los deja más vulnerables. Por lo general, el cliente cifra los datos para protegerlos en caso de un robo. Las normas de privacidad, como el GDPR mencionado anteriormente, exigen que la información se almacenen en un centro de datos en una región considerada segura.

En última instancia, la mayoría de los ataques a la seguridad comienzan en un nivel alto, como enviar un correo electrónico con malware a un empleado desprevenido. No importa si lo ejecuta de forma local o en la nube. Pero un intruso malicioso que se apodere de la cuenta de un empleado no llegará mucho más lejos a menos que pueda aprovechar las vulnerabilidades de sus servidores; Ahora bien, debe quedar claro que al ejecutar en la nube hace mucha diferencia porque la mayoría de las vulnerabilidades se encuentran en el software y no en el servicio de la nube.

Finalmente, considere sus costos de ancho de banda y redes. Sus clientes, y probablemente su personal, se comunican con servidores que pueden estar a cientos de kilómetros de distancia. Si la conexión de red no es confiable o es lenta, el rendimiento de los servidores en la nube será peor que el de su centro de datos local. Pero hoy en día, todo el mundo se conecta con trabajadores remotos, servicios SaaS y otros sistemas que están geográficamente alejados. El rendimiento de su red afectará casi todo lo que haga, ya sea que esté o no en la nube.

Ejercicios guiados

1. ¿Por qué una computadora física en un centro de nube se usa de manera más eficiente que una computadora en un centro de datos local tradicional?

2. ¿Qué es una nube híbrida?

3. ¿Qué tipo de computación en la nube requiere con mayor frecuencia el trabajo de un administrador del sistema en el lado del cliente?

4. ¿Cómo debería proteger su servicio para que no falle si utiliza un proveedor de nube?

Ejercicios de exploración

1. Compare los diferentes tipos de costos que experimenta al ejecutar sus servidores en la nube con los costos de ejecutarlos localmente.

2. Opera desde Oriente Medio, pero tiene muchos clientes en Europa y el Lejano Oriente. Describa dónde colocaría sus servicios en una oferta en la nube.

Resumen

Esta lección describió cómo funciona la computación en la nube y las ventajas y desventajas de usar la nube versus ejecutar sistemas en su propio centro de datos local. Aprendió diferentes modelos de negocios y costos, incluidas las diferencias entre nubes públicas, privadas e híbridas. Y también aprendió los diferentes tipos principales de ofertas de nube y para qué se utiliza cada una.

Respuestas a ejercicios guiados

1. ¿Por qué una computadora física en un centro de nube se usa de manera más eficiente que una computadora en un centro de datos local tradicional?

   En la nube, cada computadora puede ejecutar múltiples instancias de sistemas operativos e incluso ejecutar instancias cargadas por diferentes clientes. Por lo tanto, la computadora se utiliza con más frecuencia.

2. ¿Qué es una nube híbrida?

   Una nube híbrida utiliza tanto centros de datos de un proveedor de nube como uno o más centros de datos locales.

3. ¿Qué tipo de computación en la nube requiere con mayor frecuencia el trabajo de un administrador del sistema en el lado del cliente?

   La infraestructura como servicio (IaaS) requiere que el cliente realice la administración del sistema para tareas como la creación y carga de instancias del sistema operativo y las aplicaciones.

4. ¿Cómo debería proteger su servicio para que no caiga si utiliza un proveedor de nube?

   Elija varias zonas en cada región donde ejecute su servicio, porque es muy poco probable que muchas zonas fallen simultáneamente.

Respuestas a los ejercicios de exploración

1. Compare los diferentes tipos de costos que experimenta al ejecutar sus servidores en la nube con los costos de ejecutarlos localmente.

   En una nube, usted paga por el uso de su CPU y el almacenamiento de datos por cada período medido por el proveedor. Pero no paga ningún costo de hardware. En las instalaciones, usted tiene el costo fijo del hardware, junto con otros equipos como el aire acondicionado, más costos recurrentes como la energía y el mantenimiento físico.

2. Opera desde Oriente Medio, pero tiene muchos clientes en Europa y el Lejano Oriente. Describa dónde colocaría sus servicios en una oferta en la nube.

   Utilice una región de Medio Oriente para sus propias oficinas y clientes de Medio Oriente. Una región de Europa es importante para cumplir con el Reglamento General de Protección de Datos (GDPR). Es posible que necesite una región en China para cumplir con la Ley de Protección de Información Personal (PIPL) de China. En cualquier caso, tener regiones del Lejano Oriente y Europa es valioso para un mejor desempeño al interactuar con los clientes en esos lugares.

   Dentro de cada región, elija varias zonas para protegerse contra el fallo de una sola zona.

Linux Professional Institute

Tema 052: Licencias de software de código abierto

052.1 Conceptos de licencias de software de código abierto

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 052.1

Peso

3

Áreas de conocimiento clave

- Comprender las definiciones de software de código abierto y software libre
- Conocimiento de otros tipos de software monetariamente gratuito
- Conciencia de eventos importantes en la historia del código abierto
- Comprender qué es una licencia y qué derechos gestionan habitualmente las licencias
- Comprender cómo se puede utilizar el software existente para crear trabajos derivados
- Comprender las compatibilidades e incompatibilidades de licencias
- Comprender las licencias duales y las licencias condicionales
- Comprender las consecuencias de las infracciones de licencia
- Comprender los principios de la ley de derechos de autor y la ley de patentes; y cómo se ven afectados por las licencias de software de código abierto

Lista parcial de archivos, términos y utilidades

- Definición de software libre de la Free Software Foundation (FSF)
- Definición de software de código abierto de la Open Source Initiative (OSI)
- Licencias
- Contratos
- Software de dominio público

- software gratuito
- Shareware
- Administradores de licencias
- Permisos para usar, modificar y distribuir código y software
- Trabajos derivados y reutilización de código
- Software de código cerrado/propietario
- Distribución paga
- Distribución de software modificado y no modificado
- Software de hosting como servicio pago
- Compatibilidad de licencia
- Licencias duales y múltiples
- Licencia condicional
- Patentes de software
- Concesiones de licencias de patentes explícitas e implícitas

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	052 Licencias de software de código abierto
Objectivo:	052.1 Conceptos de licencias de software de código abierto
Lección:	1 de 1

Introducción

Cualquier desarrollador de software aprecia cuando un problema ya se ha resuelto de manera eficiente. Si la solución está disponible en línea, el reflejo comprensible es probarla: copiar o vincular el código fuente existente a la propia base de código, probarlo y dejarlo ahí si funciona, y luego olvidarse de este.

Pero conviene cierta precaución. La mayoría de las veces, el código fuente del software disponible en Internet está cubierto por una *licencia* gratuita y de código abierto (FOSS). Este capítulo analizará algunos conceptos básicos de FOSS desde un punto de vista legal y por qué es una buena idea no dar por sentado el código fuente de FOSS, sino observar detalladamente las condiciones de la licencia.

Como base para comprender sus obligaciones legales en materia de software, comprenda que una licencia rige casi todo el software. Una licencia es una especie de contrato. Todo el software, incluidos los sitios web, debe distribuirse con una versión escrita de su licencia (a menudo denominada "términos y condiciones"). Algunos programas y sitios web te obligan a marcar una casilla que indica que has leído sus términos y condiciones (y deberías hacerlo, aunque la mayoría

de la gente nunca lo hace). En cualquier caso, usted acepta implícitamente la licencia al utilizar el software.

Definiciones de software de código abierto y software libre

El software libre y de código abierto existe desde hace bastante tiempo y los términos a menudo se ven juntos. Sin embargo, existe cierta discrepancia entre lo que algunos declaran ser "software libre" y el significado formal de *software libre*, así como *software de código abierto*. Spoiler: “Código abierto” no significa simplemente que cualquiera pueda ver el código fuente; eso sería software “Fuente disponible”. El software libre y de código abierto es mucho más que eso. El software libre y de código abierto se contrasta con otros tipos que se consideran *propietarios* o de *código cerrado* porque no ofrecen todas las libertades analizadas en esta lección.

Quizás la definición de software libre más citada sea:

> Piense en “libertad de expresión”, no en “cerveza gratis”.

Analizaremos esa declaración en las siguientes secciones.

Libre como en el habla: Verdadera libertad para los usuarios

Por un lado, los desarrolladores pueden decidir simplemente renunciar a las recompensas y dificultades de vender su software y simplemente regalarlo sin compensación. En inglés este software es “gratuito” en el sentido de que nadie tiene que pagar por él, pero es gratuito sólo como la cerveza que se regala. Esta distribución gratuita no dice nada sobre otras condiciones de licencia y deja al usuario con la obligación de mirar siempre más de cerca: es posible que, por ejemplo, no se le hayan concedido los derechos necesarios para modificar o distribuir más el software.

Por otro lado, los desarrolladores pueden optar por hacer que su software sea “software libre” en el sentido de Stallman. Este término (adoptado en la década de 1980) se refiere a las libertades esenciales del usuario que se resumen a continuación:

1. Para ejecutar el software
2. Para estudiarlo
3. Para dárselo a otros (*redistribuirlo*)
4. Redistribuir copias de las versiones modificadas.

El “software libre” en esta definición es defendido por la Free Software Foundation (FSF), que introdujo el término *copyleft* (que no tiene significado legal, pero presenta una consideración

filosófica) que caracteriza la licencia de software libre.

Software de código abierto

A partir del movimiento del software libre, los defensores del código abierto surgieron a finales de los años 1990 como una forma de hacer que el software libre fuera más fácil de comprender y más popular entre las personas ajenas al movimiento. En 1998 se fundó una fundación sin fines de lucro, la *Open Source Initiative* (OSI), y Linus Torvalds, autor inicial del kernel de Linux, dio su apoyo al concepto. La Iniciativa de Código Abierto formalizó la *Definición de Código Abierto* para incluir los siguientes criterios:

1. *Redistribución gratuita*: Uno es libre de decidir cómo redistribuir el programa, ya sea de forma gratuita o para la venta, siempre que no se exijan regalías ni derechos de licencia. Esta libertad incluye incorporar el programa a otro programa.
2. *Disponibilidad del código fuente*, ya sea en línea o proporcionado junto con el software.
3. Permitir la creación y distribución de *obras derivadas* y modificaciones.
4. *Integridad del código fuente del autor*: Las modificaciones pueden restringirse si a los destinatarios se les permite modificar el programa mediante parches en el momento de la compilación y distribuir estos parches junto con el código fuente. La distribución de software *construido* a partir de código fuente modificado no puede estar restringida.
5. *Sin discriminación contra personas o grupos*: Por ejemplo, una licencia con permiso para el uso del software por parte de "sólo profesores" no cumpliría la definición de código abierto.
6. *Sin discriminación en cuanto a campos de actividad*: Por ejemplo, no restringir el uso comercial.
7. *Distribución de licencia*: Todos los que reciben el programa tienen la misma licencia original.
8. *La licencia no debe ser específica de un producto*.
9. *La licencia no debe restringir otro software*: Por ejemplo, otro software incluido con este software podría tener una licencia diferente.
10. *La licencia debe ser tecnológicamente neutra*.

Libertad versus código abierto

La Free Software Foundation no respaldó el término “Código abierto” (Open Source en Inglés), insistiendo en que oculta el objetivo clave de la libertad. Así que si bien los defensores del software libre y del software de código abierto parecen perseguir y defender el mismo concepto, los movimientos tienen motivaciones distintas. Dicho de forma simplificada, los defensores del software libre enfatizan los derechos de los desarrolladores y usuarios, mientras que los defensores del código abierto priorizan el uso generalizado y el éxito del software.

Prácticamente todo el software libre se considera de código abierto, mientras que hay muchas licencias que se consideran de código abierto (aprobadas por OSI) pero que no son libres según la FSF.

Debido a que las diferencias entre código libre y código abierto se refieren a objetivos y motivaciones más que al contenido de las licencias, y dado que ambos términos siguen siendo de uso frecuente, muchos defensores se refieren a ambas definiciones juntas usando las frases software libre y de código abierto (FOSS) o *free/libre y Open Source Software* (FLOSS). El término “libre” se refiere a la libertad en muchas lenguas romances.

Otros tipos de software monetariamente gratuito

Además de las amplias categorías de software libre y software propietario, existe una variedad de otras estrategias de distribución. Algunas de estas estrategias son:

Shareware

Este término generalmente se refiere al software propietario que está a la venta, pero que puede usarse de forma gratuita con funcionalidad limitada hasta que el usuario decida comprar la versión completa.

Freeware

Este término describe el software que se distribuye sin costo y sin limitación de uso, pero no necesariamente de conformidad con una definición formal de software libre. En muchos casos, el software gratuito es privado y el código fuente a menudo no se publica en absoluto.

Software fuente disponible

A veces, los desarrolladores de software propietario ponen a disposición su código fuente (para facilitar mejores informes de errores), pero imponen la adquisición de una licencia como condición para el uso del código fuente en otros proyectos. Este tipo de disponibilidad con limitaciones estrictas no debe confundirse con el software de código abierto.

Software de fuente compartida

Este término fue introducido por Microsoft en 2001, y fue cuando la compañía decidió hacer que parte de su código fuente de software estuviera disponible en línea para investigación y pruebas. No confunda esta definición con shareware o software fuente disponible.

Software de dominio público

Este es un software sobre el cual los autores han renunciado a todos los derechos de autor. Esta definición no se aplica en todas las jurisdicciones (especialmente en aquellas en las que al autor se le otorgan derechos de “droit d’auteur” o "derechos de autor", como en Francia o

Alemania). Se han introducido licencias como “Unlicense”, que deberían tener el mismo efecto. Además, el software también puede pasar al dominio público cuando la duración de los derechos de autor ha expirado.

Principios de la ley de derechos de autor y cómo se ven afectados por las licencias de software de código abierto

Primero y ante todo: Si no hay información de licencia disponible para un determinado archivo o proyecto de código fuente, no puede asumir que el archivo o proyecto carece de protección de derechos de autor. De hecho, ocurre todo lo contrario, al menos desde que la mayoría de las naciones del mundo firmaron el Convenio de Berna desde 1887.

En este tratado, las naciones firmantes acuerdan que una obra literaria o artística está protegida por derechos de autor tan pronto como existe (o en otras palabras, tan pronto como se “fija” en un soporte). Eso significa que un autor no tiene que registrarse ni solicitar derechos de autor. Todavía pueden hacerlo en algunos países, y el registro podría ser necesario para acciones de infracción en algunas jurisdicciones, incluido Estados Unidos.

Asimismo, los firmantes se comprometen a respetar los derechos de autor de cualquier autor de otro país firmante, que, a noviembre de 2022, ascendían a 181 de los 195 países del mundo.

Sin embargo, no todo lo que se ha creado está protegido por derechos de autor. Para calificarla como una *obra* protegida por derechos de autor, la creación debe satisfacer algunos criterios básicos: por ejemplo, los hechos y las ideas no pueden protegerse por derechos de autor, pero un texto que explica una idea puede protegerse si alcanza un cierto grado de *originalidad*. La medida de originalidad difiere en todo el mundo, pero en muchos casos, la barrera de protección es muy baja. Muchas jurisdicciones están considerando actualmente cuánta inteligencia artificial se puede utilizar para crear una obra que merezca originalidad y, por tanto, derechos de autor.

Dependiendo de la jurisdicción, los programas de ordenador están protegidos por derechos de autor como obras literarias: es decir, los derechos de autor no se aplican a la idea o algoritmo, sino a su implementación en el código fuente.

Los derechos de autor otorgan al autor los derechos exclusivos (entre otros) para copiar, modificar, sublicenciar, distribuir y publicar la obra. La recepción de la obra es gratuita, por lo que no se necesita licencia para leer un libro o escuchar una canción en la radio, siempre que para ello no sea necesario realizar una copia permanente.

Debido a que los derechos de autor surgen sin necesidad de registro, cualquiera que quiera copiar, modificar, sublicenciar, distribuir o publicar la obra de otro autor debe obtener permiso primero. Aquí es donde entran en juego las licencias, como contratos entre el autor de una obra y

una persona que quiere ejercer algunos de los derechos exclusivos del autor.

Las licencias FOSS se ofrecen a cualquier persona sin pagar ninguna tarifa. Cualquiera puede crear su propia licencia e intentar que la OSI o la FSF la aprueben. Pero se recomienda encarecidamente utilizar una licencia FOSS existente debido a su aceptación general y a la familiaridad que los usuarios de software expertos tienen con el contenido de la licencia y sus obligaciones. De hecho, sólo un bloque de las muchas licencias aprobadas por la FSF o la OSI son de uso común.

Principios del Derecho de Patentes

A diferencia del copyright, que no protege las ideas, las patentes protegen las invenciones (ideas) sin necesidad de que la idea esté fijada (todavía) en una máquina o proceso. Otra diferencia es que los inventores deben solicitar patentes explícitamente y registrarlas en la oficina de patentes del país donde buscan la protección.

Sin profundizar demasiado en la ley de patentes y sus requisitos, sólo señalaremos una cuestión muy central: la protección de una patente requiere (entre otros criterios) una idea con un determinado aspecto técnico. Históricamente, las ideas para una nueva receta de comida o un nuevo juego de mesa no califican para la protección de patente, mientras que las ideas para una nueva máquina de cocinar o una consola de juegos sí pueden calificar.

En el contexto de la programación informática, surge la cuestión de si el software podría estar sujeto a protección por patente. Esto depende tanto de la jurisdicción legal como de la aplicación particular: Por ejemplo, en Alemania los programas de ordenador como tales generalmente están excluidos de la protección de patentes. Sin embargo, si los programas se combinan con un objeto físico (por ejemplo, cuando el software controla el sistema de frenado automático de un automóvil), se puede solicitar protección mediante patente para la aplicación que incluye el elemento físico del freno y no el software como tal, y podría por lo tanto calificar.

En otras jurisdicciones, como Estados Unidos, se pueden conceder patentes para programas informáticos como tales, dependiendo de la evolución de la jurisprudencia.

Al conceder licencias de software libre se debe tener presente el concepto de protección de patentes. Algunas licencias (como la GPLv3) permiten patentes sobre el software FOSS al otorgar permiso explícito para el uso del software. Pero algunas licencias ni siquiera mencionan las patentes (como la licencia BSD-3-Clause) y otras las excluyen explícitamente (como las licencias Creative Commons). Sin embargo, en determinadas circunstancias, la concesión de patentes puede estar implícita en la licencia o puede leerse en la licencia.

NOTE Las diferentes licencias FOSS se tratan en lecciones posteriores.

Especialmente cuando se trata de software integrado, como el software en dispositivos de audio, se debe prestar especial atención a las patentes relacionadas con el software, ejemplo, mediante la realización de controles de patentes de los autores del software.

Contratos de licencia

Como se mencionó anteriormente, las licencias son contratos entre el autor de una obra y alguien que quiere ejercer algunos de los derechos exclusivos del autor. Algunas de las licencias FOSS más extensas, como las Licencias Públicas Generales GNU versión 2 y versión 3, incluyen disposiciones para la rescisión del contrato. Las licencias de software libre siempre incluyen concesiones de derechos con respecto a las libertades centrales. Normalmente, los siguientes derechos se indican o se pueden leer en el texto de la licencia:

> ...los derechos de usar, copiar, modificar, fusionar, publicar, distribuir, sublicenciar y/o vender copias del Software...
>
> — MIT license

Los *administradores de licencias* son personas o, en muchos casos, organizaciones como la FSF que gestionan las versiones de las licencias. Por ejemplo, la sección 9 de GPLv2 declara que la FSF es la administradora de licencias:

> La Free Software Foundation puede publicar periódicamente versiones revisadas y/o nuevas de la Licencia Pública General. Estas nuevas versiones serán similares en espíritu a la versión actual, pero pueden diferir en detalles para abordar nuevos problemas o inquietudes.
>
> — GNU General Public License version 2

Algunos administradores de licencias también publican preguntas frecuentes (FAQ) para ayudar a responder preguntas sobre las licencias. Estos pueden resultar útiles para iniciar una conversación entre el licenciatario y el licenciante.

Distribución

La distribución de software (en formato binario o de código fuente) es un aspecto central de la mayoría de las licencias FOSS, ya que el mero *uso* del software FOSS generalmente no genera ninguna obligación de licencia:

> Esta Licencia no cubre actividades distintas de la copia, distribución y modificación; están fuera de su alcance. El acto de ejecutar el Programa no está restringido...
>
> — GNU General Public License version 2

La mayoría de las obligaciones de licencia surgen únicamente con la distribución, es decir, la transferencia de una copia modificada o no modificada, ya sea en un medio tangible como un CD o mediante descarga. En algunos casos, las condiciones de licencia también se activan si el software se ejecuta en un servidor (sin distribuir el código fuente) mientras el usuario interactúa con el software.

La distribución de software ejecutable también puede requerir la entrega del código fuente, el texto de la licencia y los avisos de derechos de autor, según el tipo de licencia FOSS. Algunas licencias requieren que se incluyan avisos de modificación en el código fuente si se distribuye código modificado.

La distribución también podría desencadenar efectos copyleft; es decir, tras la distribución de software con licencia GPLv3 que ha sido modificado, es posible que el código fuente de todo el software modificado deba publicarse bajo GPLv3.

Si bien se puede vender software libre, normalmente no se pueden imponer tarifas de licencia. Esto significa, que alguien puede vender software bajo la licencia GPLv3 como simplemente un binario, pero como el código fuente está disponible (o debe ofrecerse), las personas interesadas en el software siempre pueden optar por obtener las fuentes (quizás de otra persona, de forma gratuita) y construir el software a partir de las fuentes.

Obras derivadas

Cuando un grupo de desarrolladores incorpora código en su propio trabajo del proyecto de otra persona, el resultado puede ser un trabajo *derivado*. Los detalles varían de un proyecto a otro y de una licencia a otra, y debido a que los detalles requieren cierta sofisticación en las técnicas de desarrollo de software, simplemente diremos que los desarrolladores deben asegurarse de incorporar el código de una manera que se ajuste a la licencia.

La cuestión más importante con respecto a las obras derivadas es que, para algunas licencias como la GPL, una obra derivada debe publicarse bajo la misma licencia. Estas licencias se denominan *recíprocas*.

El significado práctico inmediato de un requisito recíproco es que, si utiliza código GPL en su propio proyecto de manera que el suyo sea una derivación, debe revelar su código fuente y dejar que otros construyan productos a partir de el. Este requisito de licencia es muy deseado por algunos desarrolladores que quieren animar a más personas a utilizar licencias gratuitas. Sin embargo, la licencia reduce el atractivo de la GPL para algunos desarrolladores que potencialmente podrían utilizar código libre.

Muchas otras licencias gratuitas y de código abierto no imponen ese requisito. Estas tienden a

denominarse licencias *permisivas*.

Consecuencias de las infracciones de licencia

Si, en el momento de la distribución, no se cumplen las condiciones de la licencia (ejemplo, si el software con licencia GPLv3 distribuido como binario no va acompañado de una oferta de código fuente), se viola el contrato de licencia. Las consecuencias de las violaciones de licencia dependen de la licencia. La violación de la licencia GPLv3, puede dar lugar a la rescisión de la licencia. Cualquier acción adicional que requiera permiso del autor, ejemplo, distribución del software, entonces constituyen violaciones de derechos de autor.

Si una empresa incluye software con licencia GPLv3 en un producto y viola la licencia, se puede presentar un reclamo exigiendo a la empresa que retire sus productos. Las violaciones intencionales de los derechos de autor pueden incluso dar lugar a cargos penales.

Algunas licencias tienen cláusulas específicas que permiten al licenciatario corregir la infracción dentro de los 30 días posteriores a la notificación. Si la persona que distribuye el software logra cumplir con la licencia dentro de este período, se restablece la licencia.

Compatibilidad e incompatibilidad de licencias

Los grandes proyectos de software a menudo incluyen software bajo diferentes licencias, cada licencia especifica sus requisitos individuales. Estos proyectos pueden tropezar con obstáculos al utilizar software que requiere que su licencia se utilice en trabajos derivados. Esto se debe a que los términos de licencia de dichas licencias copyleft pueden diferir, haciéndolas incompatibles. Es posible que no sea posible publicar o distribuir un proyecto de software que integre componentes bajo diferentes licencias copyleft sin violar una de las licencias.

Algunas licencias copyleft enumeran explícitamente licencias compatibles, lo que facilita el uso de un componente con licencia copyleft bajo otra licencia copyleft.

La mayoría de las licencias permisivas son compatibles con otras licencias. Por ejemplo, los componentes con licencia MIT se pueden utilizar en un proyecto con licencia GPLv3 sin correr el riesgo de infringir la licencia. Sin embargo, un componente con licencia GPLv3 no siempre se puede utilizar en un proyecto con licencia MIT sin violar los términos de GPLv3. Por lo tanto, es posible que la compatibilidad no siempre funcione en ambos sentidos.

Antes de lanzar un proyecto de software al mercado, los desarrolladores y sus asesores legales deben realizar una verificación exhaustiva de la compatibilidad de las licencias para evitar infracciones de las mismas. La gestión del cumplimiento del código abierto debe integrarse en las primeras etapas de un proceso de desarrollo de software para evitar retrasos en la distribución

del software, por ejemplo, problemas de incompatibilidad de licencias. Busque minuciosamente en el código fuente las licencias aplicables (utilizando herramientas de escaneo de software) y verifique si se cumplen todas las condiciones de la licencia.

Licencia dual y licencias múltiples

Algunos programas pueden estar disponibles bajo varias licencias. Por ejemplo, un licenciante puede optar por una *licencia dual* para su proyecto tanto con una licencia copyleft como la GPLv3 como con una licencia propietaria. La licencia de propiedad podría ser necesaria, si el posible licenciatario incorpora el código en su propio producto de propiedad. Cada desarrollador determina si puede cumplir con las condiciones de la GPLv3 o si debe obtener la licencia propietaria, y lo más probable es que implique una tarifa de licencia.

Como se señaló anteriormente, algunos programas pueden incluir componentes bajo varias licencias con diferentes condiciones de licencia. Aunque la mayoría de las licencias no suelen causar incompatibilidad, diferentes licencias pueden tener diferentes requisitos para diferentes partes del software.

Ejercicios guiados

1. ¿Qué significa el acrónimo FOSS?

2. ¿Cuáles de las siguientes pertenecen explícitamente a las libertades esenciales del usuario del software libre?

Ejecutar software	
Software de estudio	
Software de copia	
Cambiar software	
Publicar software	
Redistribuir software	

3. ¿El código fuente que se encuentra en Internet sin información de licencia está libre de modificación y distribución por parte de cualquier persona? Por favor explique.

4. ¿Se puede patentar el software?

Sí	
No	
Depende	

5. ¿Qué es un administrador de licencias?

Lo mismo que un licenciante	
Alguien que pueda proponer versiones futuras de una licencia	
Alguien que puede rescindir una licencia	
Alguien que maneja software de aviones	

6. ¿La compatibilidad de licencias siempre funciona en ambos sentidos?

Sí, si dos licencias son compatibles, no importa si A está incluida en B o B está incluida en A.	
No, en algunos casos la licencia A puede incluirse en un proyecto bajo la licencia B, pero es posible que no se permita que B se distribuya bajo la licencia A.	
No, la compatibilidad de licencias es siempre unidireccional.	

Ejercicios de exploración

1. ¿Son compatibles la licencia GPLv2 y LGPLv2.1? Por favor dé una breve explicación.

2. ¿Se puede publicar software bajo licencias de Contenido Abierto (como CC-BY)?

Sí, las licencias CC se pueden aplicar a cualquier trabajo protegido por derechos de autor.	
No, el software sólo puede estar protegido por patentes.	
No, las licencias CC no se aplican al software.	

1. ¿Por qué es importante la distribución en el contexto de las licencias de software libre?

Resumen

Esta lección proporciona una introducción a los conceptos básicos del software libre y de código abierto, así como a los conceptos subyacentes de derechos de autor y patentes. Explica algunos de los fundamentos y la historia tanto del software libre como del software de código abierto y ayuda a distinguir ambos de los conceptos de licencia propietaria. La definición de código abierto de OSI ayuda a categorizar las licencias como licencias de código abierto.

Si bien ya existen docenas de licencias FOSS, cualquiera es libre de introducir licencias adicionales.

No se debe subestimar el concepto de licencia: las licencias otorgan permisos para manejar el software de formas que de otro modo estarían reservadas exclusivamente al autor. El incumplimiento de las licencias puede dar lugar a disputas legales. Por lo tanto, se debe buscar asesoramiento legal sobre el cumplimiento de la licencia antes de distribuir o incorporar software a otro código de proyecto.

Respuestas a ejercicios guiados

1. ¿Qué significa el acrónimo FOSS?

   Software libre y de código abierto.

2. ¿Cuáles de las siguientes pertenecen explícitamente a las libertades esenciales del usuario del software libre?

Ejecutar software	X
Software de estudio	X
Software de copia	
Cambiar software	X
Publicar software	
Redistribuir software	X

3. ¿El código fuente que se encuentra en Internet sin información de licencia está libre de modificación y distribución por parte de cualquier persona? Por favor explique.

   No. El código fuente, si alcanza el umbral de originalidad, está protegido por derechos de autor por defecto como obra literaria. Dado que la modificación y distribución constituyen derechos exclusivos del autor, nadie más que el autor puede hacerlo ni dar permiso para hacerlo.

4. ¿Se puede patentar el software?

Sí	
No	
Depende	X

5. ¿Qué es un administrador de licencias?

Lo mismo que un licenciante	
Alguien que pueda proponer versiones futuras de una licencia	X
Alguien que puede rescindir una licencia	
Alguien que maneja software de aviones	

6. ¿La compatibilidad de licencias siempre funciona en ambos sentidos?

Sí, si dos licencias son compatibles, no importa si A está incluida en B o B incluida en A.	
No, en algunos casos la licencia A puede incluirse en un proyecto bajo la licencia B, pero es posible que no se permita que B se distribuya bajo la licencia A.	X
No, la compatibilidad de licencias es siempre unidireccional.	

Respuestas a los ejercicios de exploración

1. ¿Son compatibles la licencia GPLv2 y LGPLv2.1? Por favor dé una breve explicación.

   Si el software A tiene licencia GPLv2 y el software B tiene licencia LGPLv2.1, es posible usar B en A, ya que LGPLv2.1 permite su uso bajo las condiciones de la licencia GPLv2. Sin embargo, A no se puede utilizar en B según los términos de LGPLv2.1. Si A se utiliza en B, todo el software debe tener la licencia GPLv2, lo cual es posible porque LGPLv2.1 permite el uso del software bajo GPLv2.0.

2. ¿Se puede publicar software bajo licencias de Contenido Abierto (como CC-BY)?

Sí, las licencias CC se pueden aplicar a cualquier trabajo protegido por derechos de autor.	X
No, el software sólo puede estar protegido por patentes.	
No, las licencias CC no se aplican al software.	

3. ¿Por qué es importante la distribución en el contexto de las licencias de software libre?

   Muchas obligaciones de licencia de FOSS se activan tras la distribución del software. Si el software nunca se distribuye, sino que sólo se modifica y utiliza en un sistema cerrado (como una división de una empresa), es posible utilizar el software sin cumplir con las obligaciones de licencia.

052.2 Licencias de software Copyleft

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 052.2

Peso

3

Áreas de conocimiento clave

- Comprender el concepto de licencias de software copyleft
- Comprender los derechos otorgados por las licencias de software copyleft
- Comprender las obligaciones creadas por las licencias de software copyleft
- Comprender las principales propiedades de las licencias de software copyleft comunes
- Comprender la compatibilidad de las licencias de software copyleft con otras licencias de software
- Conocimiento de los términos “licencia recíproca” y “licencia restrictiva”

Lista parcial de archivos, términos y utilidades

- Copyleft
- Distribución
- Transmisión
- Tivoization
- Licencia pública general GNU, versión 2.0 (GPLv2)
- Licencia pública general GNU, versión 3.0 (GPLv3)
- Licencia pública general reducida GNU, versión 2 (LGPLv2)
- Licencia pública general reducida GNU, versión 3 (LGPLv3)

- Licencia pública general GNU Affero, versión 3 (AGPLv3)
- Licencia pública de Eclipse (EPL), versión 1.0
- Licencia pública de Eclipse (EPL), versión 2.0
- Licencia pública de Mozilla (MPL)

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	052 Licencias de software de código abierto
Objectivo:	052.2 Licencias de software copyleft
Lección:	1 de 1

Introducción

Ya se ha descrito la importancia de las licencias tanto para el uso como para el desarrollo de software. Por lo tanto, no sorprende que el software libre también se haya caracterizado por nuevos enfoques en materia de licencias desde el principio: las condiciones para el uso sin restricciones o el desarrollo colaborativo del software deben estar legalmente definidas para poder protegerlas y hacerlas cumplir.

Conscientes de que la ley de derechos de autor, que está firmemente arraigada en casi todos los sistemas legales del mundo, no podía ser cuestionada ni reemplazada, los desarrolladores de software adoptaron ya en la década de 1980 un enfoque que respeta las normas de derechos de autor pero las complementa con nuevas regulaciones que enfatizan el principio de "` libertad`": *copyleft*.

Copyleft y la Licencia Pública General GNU (GPL)

Richard Stallman, entonces desarrollador del renombrado MIT, fundó el *Proyecto GNU* en 1983 para desarrollar lo que él consideraba un sistema operativo “libre”. Pronto quedó claro que el código desarrollado por el proyecto tenía que estar protegido legalmente para que no pudiera

simplemente ser asumido por proveedores comerciales y así convertirse en “no libre”.

Por lo tanto, Stallman fundó la *Free Software Foundation* (FSF) sin fines de lucro en 1985, que resume su misión en su sitio web de la siguiente manera: “La Free Software Foundation está trabajando para asegurar la libertad de los usuarios de computadoras promoviendo el desarrollo y el uso de software libre (como en libertad) software y documentación...”

Un instrumento clave para esta misión es una licencia que respete la ley aplicable (especialmente la ley de derechos de autor) por un lado y que implemente sus propias ideas de libertad de manera legalmente limpia, por el otro. El resultado fue la primera versión de la *Licencia Pública General GNU* (GPLv1) en 1989. Esta licencia y numerosos artículos, como “¿Qué es el software libre?”, escrito por Stallman en 1992, dejan en claro las motivaciones y valores de desarrolladores de software libre, que ahora también se ven a sí mismos como un “movimiento”.

El núcleo programático sigue formado por las “`cuatro libertades esenciales`” formuladas por Stallman en el citado artículo, cuya numeración comienza con 0:

> - La libertad de ejecutar el programa como desees para cualquier propósito (libertad 0).
> - La libertad de estudiar cómo funciona el programa y cambiarlo para que funcione como desees (libertad 1). El acceso al código fuente es una condición previa para ello.
> - La libertad de redistribuir copias para poder ayudar a otros (libertad 2).
> - La libertad de distribuir copias de sus versiones modificadas a otros (libertad 3). Al hacer esto, podrá brindarle a toda la comunidad la oportunidad de beneficiarse de sus cambios. El acceso al código fuente es una condición previa para ello.
>
> — Richard Stallman, What is Free Software?

A diferencia de las licencias para productos comerciales, que imponen restricciones de uso en primer plano, el software libre implica la máxima libertad para usuarios y desarrolladores.

El preámbulo de GNU GPLv1 lo resume de la siguiente manera:

> Específicamente, la Licencia Pública General está diseñada para garantizar que usted tenga la libertad de regalar o vender copias de software libre, que reciba el código fuente o pueda obtenerlo si lo desea, que pueda cambiar el software o usar fragmentos en nuevos programas.
>
> — GNU General Public License, version 1

Esto significa que todos tienen derecho a usar, distribuir y modificar software bajo la GPL sin restricciones (lo cual es posible porque el código fuente es accesible, es decir, “abierto”) y, a su vez, distribuir las modificaciones. Incluso es posible cobrar dinero por transmitir el software:

> De hecho, animamos a las personas que redistribuyen software libre a cobrar todo lo que quieran o puedan. Si una licencia no permite a los usuarios hacer copias y venderlas, es una licencia no libre... Los programas libres a veces se distribuyen gratuitamente y otras veces por un precio sustancial. A menudo, el mismo programa está disponible en ambos sentidos desde diferentes lugares. El programa es libre independientemente del precio, porque los usuarios tienen libertad para utilizarlo.
>
> — Free Software Foundation, Selling Free Software

Pero si las libertades son de tan amplio alcance, ¿hasta qué punto está protegido el software bajo esta licencia ejemplo, contra su incorporación a productos propietarios?

Esta es la función del principio copyleft mencionado anteriormente, que la GPL ya aplica en la versión 1 aunque el término aún no aparezca explícitamente:

> No puede copiar, modificar, sublicenciar, distribuir o transferir el Programa excepto según lo expresamente dispuesto en esta Licencia Pública General.
>
> — GNU General Public License version 1

Esto significa que todas estas libertades están vinculadas a la condición de que los usuarios las preserven en todo lo que hacen con el software.

Por lo tanto, el copyleft no sólo garantiza libertades, sino que también exige que todos los usuarios concedan estas libertades a otros. Esto se logra estipulando que el software bajo una licencia copyleft (como la primera GPLv1) puede modificarse y redistribuirse sólo si las cambios se publican en las mismas condiciones, es decir, bajo la misma licencia.

Por lo tanto, el ideal del software libre, es decir, el uso colectivo y el desarrollo posterior del software, tiene prioridad sobre las necesidades personales que los individuos puedan tener en relación con el software. El principio de reciprocidad es crucial: quienes utilizan las libertades también deben otorgarlas. Por lo tanto, las licencias copyleft a menudo se denominan *recíprocas*.

Este enfoque completamente nuevo de una licencia de software ya demostró ser legalmente sólido y practicable en la versión 1 de la GPL, por lo que la GPL sólo ha pasado por dos revisiones importantes en los casi 40 años durante los cuales se ha desarrollado el mercado moderno de TI.

GPLv2 y GPLv3

En 1991, la Free Software Foundation presentó la versión 2 de la Licencia Pública General GNU (GPLv2), que se estableció durante muchos años como la licencia más popular para proyectos de software libre. Por ejemplo, el núcleo del sistema operativo Linux todavía tiene hoy la licencia GPLv2.

En comparación con la versión 1, la versión 2 se ocupa principalmente de definiciones más precisas para evitar ambigüedades. Por ejemplo, la versión 2 explica con mucho más detalle lo que se entiende por “código fuente”.

También es interesante el nuevo artículo 7, que establece el principio de libertad y, por tanto, la validez de la licencia como absoluto y no permite ningún compromiso, por ejemplo, la integración de partes menos libres en el software:

> Si no puede distribuir para satisfacer simultáneamente sus obligaciones bajo esta Licencia y cualquier otra obligación pertinente, entonces, como consecuencia, no podrá distribuir el Programa en absoluto. Por ejemplo, si una licencia de patente no permitiera la redistribución libre de regalías del Programa por parte de todos aquellos que reciben copias directa o indirectamente a través de usted, entonces la única manera de satisfacer tanto dicha licencia como esta Licencia sería abstenerse por completo de distribuir la patente del Programa.
>
> — GNU General Public License version 2

No fue hasta 16 años después, en 2007, que la FSF publicó una nueva versión de la GPL para tener en cuenta las innovaciones técnicas — como la prestación de servicios de software a través de Internet — así como cuestiones de compatibilidad con otras licencias FOSS. Sin embargo, la licencia se mantiene estable en términos de sus declaraciones principales y simplemente agrega detalles para mayor aclaración. Echemos un vistazo más de cerca a algunas de estas adiciones.

Mientras que la GPLv2 todavía se refiere generalmente al suministro de software como *distribución*, la GPLv3 especifica este proceso con dos nuevos términos: *propagación* y *transmisión*. La razón principal de esto es que el término "distribución" está definido en numerosas leyes de derechos de autor en todo el mundo. Para evitar ambigüedades o conflictos, la GPLv3 elige estos nuevos términos y los define de la siguiente manera:

> “Propagar” significa hacer cualquier modificación en un trabajo sin permiso, esto lo haría responsable directa o secundariamente de una infracción de las leyes de derechos de autor aplicables, excepto ejecutarla en una computadora o modificar una copia privada. La propagación incluye la copia, la distribución (con o sin modificación), la puesta a disposición del público y, en algunos países, también otras actividades.
>
> “Transmitir” un trabajo significa cualquier tipo de propagación que permita a otras partes realizar o recibir copias. La mera interacción con un usuario a través de una red informática, sin transferencia de copia, no constituye transmisión.
>
> — GNU General Public License version 3

Ante el número significativamente creciente de productos de software comercial cuya distribución está restringida por los fabricantes mediante medidas técnicas como códigos de

registro o componentes de hardware (los llamados *dongles*), a finales de los años 1990 hubo una serie de iniciativas legales internacionales para tipificar como delito la elusión de estas medidas. La llamada *gestión de derechos digitales* (DRM - siglas en Inglés), a la que sus opositores también llaman despectivamente *gestión de restricciones digitales*, es una espina clavada para la FSF, ya que las medidas contradicen fundamentalmente la exigencia de la libre distribución de software.

En respuesta, la versión 3 de la GPL contiene un pasaje que establece que el software bajo la GPL no puede modificarse con referencia a los requisitos legales de DRM. Esto también significa que el software con licencia GPLv3 puede utilizar DRM, pero que otros también pueden eludir dichas medidas.

El término *tivoización* también se utiliza frecuentemente en este contexto. La palabra apareció explícitamente en los primeros borradores de GPLv3, pero no se incluyó en la versión final. El término se remonta a la empresa TiVo, que utilizaba software con licencia GPLv2 en su grabador de vídeo digital, pero al mismo tiempo impedía técnicamente la instalación y el uso de software modificado en el dispositivo. En opinión de la FSF, esto contradecía los principios de la GPL y, después de algunas discusiones, la GPLv3 lo tiene en cuenta con un párrafo sobre los llamados *productos de usuario*. Generalmente estipula que los productos que utilizan software con licencia GPLv3 también deben proporcionar información sobre cómo se puede modificar este software.

Otro añadido se refiere a las patentes, que la FSF rechaza fundamentalmente en el caso del software, por considerar que obstaculizan la libertad y la innovación. Esto ya se indica en el preámbulo de la GPLv3:

> Por último, cada programa está constantemente amenazado por las patentes de software. Los Estados no deberían permitir que las patentes restrinjan el desarrollo y uso de software en computadoras de uso general, pero en aquellos que lo hacen, queremos evitar el peligro especial de que las patentes aplicadas a un programa libre puedan convertirlo efectivamente en propietario. Para evitar esto, la GPL garantiza que no se pueden utilizar patentes para convertir el programa en "no libre".
>
> — GNU General Public License version 3

El texto de la licencia también contiene varios pasajes que permiten la inclusión de código bajo una patente mediante una “licencia de patente no exclusiva, mundial y libre de regalías” del licenciante con el fin de proteger a los usuarios de dicho código de disputas entre patentes, titulares y licenciatarios.

La Licencia Pública General GNU Affero (AGPL)

Con la creciente disponibilidad y velocidad de Internet, están surgiendo cada vez más servicios en los que el software simplemente se instala en los servidores del proveedor, — el *Application*

Service Provider (ASP) --, y cuyos clientes interactúan con los servicios a través de Internet. La tendencia ha ganado el nombre de *Software como servicio* (SaaS).

En tales casos, la GPLv2 no proporcionó ninguna claridad sobre si el código fuente (posiblemente modificado por el proveedor) debería estar disponible y cómo. La versión 3 de la GPL cierra esta laguna jurídica (loophole), conocida como la *laguna ASP*, al hacer referencia explícita en la sección 13 a otra licencia emitida por la FSF en 2007: la *Licencia pública general GNU Affero* versión 3 (GNU AGPLv3). El nombre se remonta a la empresa Affero, que desarrolló y publicó las dos primeras versiones de esta licencia.

Esta AGPLv3 corresponde básicamente a la GPLv3, pero regula explícitamente el problema ASP en el apartado “interacción de red remota”. Además, ambas licencias establecen explícitamente que se pueden combinar entre sí sin restricciones.

En resumen, GNU AGPL es un suplemento complementario de la GPL. La AGPL amplía el alcance de la GPL aplicando el copyleft al software que ya no se utiliza en instalaciones locales, sino exclusivamente en forma de servicios transmitidos a través de redes.

Compatibilidad de licencias Copyleft

El desarrollo de software libre prospera basándose en el trabajo conjunto, es decir, integrando, modificando y compartiendo el código fuente de otros. Si todas las partes de un software modificado o recién compilado están bajo la misma licencia copyleft, como: GPLv3, esto no tendrá ningún problema legal. Ya que la licencia simplemente requiere que el resultado se distribuya bajo la misma licencia.

Las cosas se vuelven más complicadas cuando el software consta de partes que tienen licencias diferentes. Aquí es necesario tener en cuenta varios factores.

Obras Combinadas y Derivadas

A veces el software libre se crea en condiciones muy diferentes. Los cambios van desde simples correcciones de errores hasta proyectos complejos con millones de líneas de código. Independientemente del alcance, se hace una distinción básica entre dos tipos de trabajos cuando se trata de licencias: *derivadas* y *combinadas*.

Supongamos, que al proyecto de software A le falta una determinada funcionalidad. En lugar de desarrollar esta funcionalidad desde cero, tiene sentido combinar el código de otro, por ejmeplo el proyecto B, que ofrece exactamente esta funcionalidad. El software de B ni siquiera tendría que cambiarse para esto, sino que simplemente podría agregarse a A. Por tanto, se trata, de una obra combinada. Si tanto A como B están bajo la misma licencia copyleft, no hay problemas para el

trabajo combinado.

Si A y B están bajo diferentes licencias copyleft, se requiere precaución: ¿la combinación de A y B ya es un trabajo separado? Y, en caso afirmativo, ¿bajo qué licencia se puede o se debe licenciar? ¿O se puede evitar un conflicto asegurando que ambas partes A y B permanezcan separadas con sus respectivas licencias y no constituyan una obra nueva?

Se vuelve aún más difícil con trabajos derivados, es decir, cuando el proyecto A puede hacer uso de la funcionalidad de B, sólo incorporando su código directamente en el de A. Esta integración crea un nuevo trabajo derivado cuyas partes ya no se pueden separar.

El copyleft entra en juego para una obra derivada. Por ejemplo, A está bajo una licencia copyleft como la GPLv3, el nuevo trabajo derivado también debe estar bajo la GPLv3 de acuerdo con el principio de reciprocidad. Pero ¿qué pasa si B tiene una licencia diferente? ¿Es una licencia copyleft y es compatible con GPLv3? O, si se trata de un tipo diferente de licencia, ¿B también puede publicarse bajo una licencia diferente, es decir, volver a obtener la licencia? ¿O las licencias de A y B son categóricamente excluyentes entre sí?

El objetivo de esta lección no es enumerar las posibles combinaciones y posibles soluciones. Lo importante es ilustrar la complejidad del problema resultante de la combinación de cuestiones muy diferentes.

Técnicamente, lo primero que hay que aclarar es cómo funcionan juntas las diferentes partes del software (en nuestro caso A y B): ¿Se pueden separar entre sí o hay procesos cuya ejecución ya no se puede asignar claramente a una parte o a la otra?

Desde un punto de vista jurídico, surgen dudas sobre la relación entre las licencias de A y B. ¿Son compatibles entre sí, o sólo en partes o sólo en una dirección? ¿Volver a obtener la licencia es una opción?

Aquí sólo podemos insinuar la complejidad de estas cuestiones. Estas decisiones requieren sólidos conocimientos jurídicos. Por lo tanto, las decisiones sobre licencias para obras nuevas, pero especialmente para obras combinadas y derivadas, deben tomarse *pronto*, *cuidadosamente* y siempre después de un *asesoramiento legal* detallado.

Copyleft más débil

El copyleft ha demostrado en las últimas décadas ser extremadamente robusto, especialmente en las versiones 2 y 3 de la GPL. Sin embargo, las exigencias técnicas han llevado a la FSF a reaccionar adaptando sus licencias. La Licencia Pública General GNU Affero es un ejemplo de ello. Discutimos otros ejemplos en las siguientes subsecciones.

La Licencia Pública General Reducida (LGPL) de GNU

Un método utilizado frecuentemente en el desarrollo de software es el uso de pequeños módulos para tareas estándar, como abrir o guardar archivos. Estos módulos, o colecciones de dichos módulos, se denominan *bibliotecas o librerías de software*. Por lo general, no son aplicaciones ejecutables de forma independiente, sino rutinas que la aplicación real integra según sea necesario. El proceso de integración se conoce como *linking* y se distingue entre dos métodos.

Con las *bibliotecas estáticas*, la aplicación real (a través de programas auxiliares y pasos intermedios) integra firmemente el código de los módulos en el archivo ejecutable de la aplicación. Por otro lado, con las *bibliotecas dinámicas*, la aplicación integra un módulo solo cuando es necesario en tiempo de ejecución y lo carga en la memoria de trabajo.

Esto plantea una pregunta con respecto al copyleft y las licencias: ¿Cuáles son las implicaciones de la vinculación para las licencias? Por ejemplo, ¿esta forma de asumir el código requiere que una aplicación que utilice una biblioteca bajo la GPL adopte la propia GPL? ¿Y esto se aplica tanto a los enlaces estáticos como a los dinámicos?

El proceso de vinculación es tan omnipresente en el desarrollo de software, y los problemas asociados con el copyleft recíproco son tan extensos, que la FSF buscó desde el principio una solución de licencia que permitiera la vinculación a través de la *Licencia pública general reducida GNU* (LGPL). La LGPL se actualiza al mismo tiempo que cada nueva versión de la GPL. El nombre de la licencia en la versión 1 era *Licencia pública general de biblioteca GNU*, lo que reveló el problema original que llevó a la creación de la licencia. En las versiones 2 y 3, “Biblioteca” se reemplazó por “Lesser” para indicar lo que realmente está en juego, es decir, un debilitamiento pragmático del principio copyleft.

Por lo tanto, el software puede utilizar una biblioteca con licencia LGPL sin que este software esté sujeto automáticamente al copyleft. Los proyectos de software bajo las llamadas licencias *permisivas* (tratadas en otra lección), en particular, se benefician de este compromiso, ya que crea protección legal para la combinación de enfoques que no son compatibles bajo la ley de licencias.

Cualquier cambio en el software con licencia LGPL todavía está sujeto al copyleft, es decir, también debe estar bajo la LGPL.

Sin embargo, la biblioteca con licencia LGPL debe ser intercambiable para permitir al usuario del software reemplazar la biblioteca con una versión modificada. Para ello deben crearse las condiciones adecuadas, es decir, debe informarse sobre cómo se puede realizar dicha sustitución.

Otras licencias Copyleft

Otros proyectos y organizaciones de FOSS también buscan el mejor marco legal para sus

necesidades y, por tanto, desarrollan sus propias licencias. Un ejemplo es la *Fundación Mozilla*, fundada en 1998 y hoy más conocida por los dos proyectos que apoya: el navegador de Internet Firefox y el cliente de correo electrónico Thunderbird.

La versión 1 de la *Licencia pública de Mozilla* (MPL) se publicó en 1998 y la versión 2.0 actual (a partir de 2024) en 2012.

Al igual que la LGPL, la MPL a menudo se denomina licencia "copyleft débil". De hecho, busca lograr un equilibrio entre los estrictos requisitos del copyleft y las posibilidades de integración con productos comerciales. Esto lo logra, entre otras cosas, a través de un principio conocido como *copyleft a nivel de archivo*: si realiza un cambio en un archivo que pertenece al software bajo MPL, puede integrar este archivo en software propietario siempre que el archivo modificado sea permanece bajo la MPL y por lo tanto es accesible.

Otro ejemplo de una licencia copyleft débil es la *Eclipse Public License* (EPL) de la Fundación Eclipse. La versión actual 2.0 de 2017 es muy similar a la MPL y a menudo se la conoce como la licencia copyleft más “favorable para los negocios”. Sin embargo, las diversas licencias copyleft (y hay otras además de las mencionadas aquí) a menudo surgieron debido a desarrollos históricos más que a diferencias legales claras.

Ejercicios guiados

1. ¿Qué significa la abreviatura GPL?

2. ¿Por qué las licencias copyleft también se denominan recíprocas?

3. ¿Qué licencia copyleft FSF es adecuada para bibliotecas de software?

4. ¿Qué términos en inglés reemplazan el término “distribución” en GPLv3 y por qué?

5. ¿Cuál de las siguientes licencias copyleft fueron emitidas por la Free Software Foundation?

GPL version 3	
AGPL version 1	
LGPL version 2	
MPL 2.0	
EPL version 2	

Ejercicios de exploración

1. ¿Cuáles de las siguientes son licencias copyleft?

GPL version 2	
3-clause BSD License	
LGPL version 3	
CC BY-ND	
EPL version 2	

2. ¿Se puede normalmente crear un trabajo derivado combinando partes de dos proyectos de software que están bajo diferentes licencias copyleft fuertes? ¡Dar razones!

3. ¿Cuáles de las siguientes licencias tienen un copyleft fuerte y cuáles tienen un copyleft débil?

Common Development and Distribution License (CDDL) 1.1	
GNU AGPLv3	
Microsoft Reciprocal License (MS-RL)	
IBM Public License (IPL) 1.0	
Sleepycat License	

4. Describa algunos problemas de compatibilidad que podrían surgir al combinar software con una licencia copyleft débil con software sin licencia copyleft.

Resumen

Esta lección trata de las características de las licencias de software que siguen el principio del copyleft. Desarrollada en la década de 1980 por la Free Software Foundation, la Licencia Pública General GNU (GPL) es actualmente la licencia más popular con fuertes derechos de autor. A pesar de varias revisiones hasta la actual versión 3, sus requisitos básicos se han mantenido prácticamente sin cambios: las libertades otorgadas por la licencia para usar, redistribuir y modificar el software sin restricciones deben preservarse en todo momento. Esto significa que el software modificado sólo podrá distribuirse bajo las mismas condiciones (es decir, la misma licencia).

Las innovaciones técnicas, así como el deseo de un margen de maniobra legal a la hora de colaborar con otros proyectos, han llevado a la FSF a desarrollar licencias complementarias o alternativas. Por ejemplo, la Licencia Pública General Reducida (LGPL) de GNU, tiene en cuenta la vinculación estática o dinámica de bibliotecas de software utilizadas frecuentemente en el desarrollo de software. La Licencia Pública General GNU Affero (AGPL) responde a la tendencia técnica hacia el uso de software en forma de servicios a través de la red (especialmente Internet), es decir, no en instalaciones locales.

Además de la FSF, otros proyectos como la Fundación Mozilla y la Fundación Eclipse han desarrollado licencias copyleft. Estos también buscan un compromiso entre asegurar las libertades otorgadas por la licencia y una relación más simple con el software bajo otras mediante un copyleft más débil.

En principio, la compatibilidad de las licencias es una cuestión importante en las licencias copyleft, porque el principio del desarrollo de software libre y colaborativo debe conciliarse con las condiciones legales determinadas por la licencia respectiva. En cualquier forma de combinación de software bajo diferentes licencias, las condiciones legales deben examinarse cuidadosamente en cada caso individual.

Respuestas a ejercicios guiados

1. ¿Qué significa la abreviatura GPL?

   General Public License

2. ¿Por qué las licencias copyleft también se denominan recíprocas?

   El principio de copyleft exige que las libertades otorgadas por una licencia también se concedan a otros sin restricciones. Por ejemplo, si realiza un cambio en el software bajo la GPL, está obligado a poner esos cambios a disposición de otros en las mismas condiciones, de acuerdo con esta reciprocidad.

3. ¿Qué licencia copyleft FSF es adecuada para bibliotecas de software?

   GNU Lesser General Public License (GNU LGPL)

4. ¿Qué términos en inglés reemplazan el término “distribución” en GPLv3 y por qué?

   Los términos “transmitir” y “propagar” reemplazan a “distribución”. El trasfondo de esto es que el término “distribución” está firmemente arraigado en la ley internacional de derechos de autor. Para evitar conflictos o malentendidos, la GPL en su versión 3 no utiliza este término.

5. ¿Cuál de las siguientes licencias copyleft fueron emitidas por la Free Software Foundation?

GPL version 3	X
AGPL version 1	
LGPL version 2	X
MPL 2.0	
EPL version 2	

Respuestas a los ejercicios de exploración

1. ¿Cuáles de las siguientes son licencias copyleft?

GPL version 2	X
3-clause BSD License	
LGPL version 3	X
CC BY-ND	
EPL version 2	X

2. ¿Se puede normalmente crear un trabajo derivado combinando partes de dos proyectos de software que están bajo diferentes licencias copyleft fuertes? ¡Dar razones!

   No. Las licencias con un copyleft fuerte generalmente requieren que las versiones modificadas estén bajo la misma licencia. Esto también excluye la renovación de licencias. La combinación de licencias, cuando ambas siguen estos principios, representa por tanto una contradicción irresoluble.

3. ¿Cuáles de las siguientes licencias tienen un copyleft fuerte y cuáles tienen un copyleft débil?

Common Development and Distribution License (CDDL) 1.1	weak
GNU AGPLv3	strong
Microsoft Reciprocal License (MS-RL)	weak
IBM Public License (IPL) 1.0	weak
Sleepycat License	strong

4. Describa algunos problemas de compatibilidad que podrían surgir al combinar software con una licencia copyleft débil con software sin licencia copyleft.

   Mientras que un copyleft fuerte requiere que el software modificado se distribuya bajo la misma licencia, las condiciones son “relajadas” en el caso de un copyleft débil. Sin embargo, cualquier combinación con otras licencias plantea cuestiones fundamentales de compatibilidad. Es necesario tener en cuenta aspectos importantes a la hora de responder a estas preguntas: ¿Las partes originales de los dos proyectos de software están claramente separadas en el nuevo trabajo y, si es necesario, se pueden seguir licenciando de forma diferente? ¿A qué nivel se produce realmente la conexión entre los dos proyectos de software originales? ¿Se adopta directamente el código fuente o “solo” está vinculado dinámicamente

con el otro software (biblioteca)? ¿Una de las dos licencias generalmente permite volver a obtener la licencia? Todas las preguntas de este tipo sólo pueden responderse tras un análisis preciso de las respectivas licencias y con conocimientos jurídicos especializados.

052.3 Permissive Software Licenses

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 052.3

Peso

3

Áreas de conocimiento clave

- Comprender el concepto de licencias de software permisivas
- Comprender los derechos otorgados por licencias de software permisivas
- Comprender las obligaciones creadas por las licencias de software permisivas
- Comprender las principales propiedades de las licencias de software permisivas comunes
- Comprender la compatibilidad de las licencias de software permisivas con otras licencias

Lista parcial de archivos, términos y utilidades

- Licencia BSD de 2 cláusulas
- Licencia BSD de 3 cláusulas
- Licencia MIT
- Licencia Apache, versión 2.0

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	052 Licencias de software de código abierto
Objectivo:	052.3 Licencias de software permisivas
Lección:	1 de 1

Introducción

Las licencias *permisivas* son actualmente las licencias de código abierto más utilizadas, a diferencia de las licencias *restrictivas* como la Licencia Pública General GNU (GPL). Las licencias de software permisivas tienden a ser simples y flexibles, y brindan una amplia gama de libertad a sus autores, quizás la libertad más amplia disponible entre las licencias de código abierto.

Este tipo de licencia otorga amplia libertad a los desarrolladores de software con respecto al uso, modificación y redistribución del software, siempre que reconozcan al autor original. Por ejemplo, normalmente alguien puede distribuir una obra derivada bajo una licencia de código cerrado, siempre que la obra incluya la atribución del autor de la obra de la que se derivó la nueva.

El principio detrás de esta lógica es permitir la máxima difusión posible del software. Las licencias permisivas pretenden beneficiar a los individuos y al público, ya que facilita el uso comercial del software, preservando al mismo tiempo los derechos del autor original mediante la atribución

No es casualidad que las primeras y más importantes licencias de software permisivo se desarrollaron en el ámbito académico, como las licencias tipo BSD de la Universidad de Berkeley

en California o la Licencia MIT/X11 del Instituto Tecnológico de Massachusetts. Estas se conocen como licencias académicas de código abierto. Su influencia dentro de la industria fue tal que sentaron las bases para licencias de software permisivas similares concebidas fuera de contextos académicos, como la Licencia Apache de la Apache Software Foundation.

Derechos y obligaciones de las licencias de software permisivas

En general, las licencias de software permisivas más populares otorgan al licenciatario el derecho ilimitado a:

Utilice el software

Cualquier persona (desde usuarios individuales hasta empresas comerciales y autoridades públicas) puede utilizar el software cubierto por una licencia de software permisiva y para cualquier fin, ya sea personal o profesional.

Modificar el software

El software se puede mejorar, adaptar o incluso integrar como un componente (por ejemplo, una biblioteca) en otro software.

Redistribuir el software

Si el software se modifica, tomando la forma de una obra derivada, se puede redistribuir bajo diferentes licencias, incluidas las propietarias.

La única obligación en las licencias de software permisivas suele ser la de atribución: El licenciatario debe indicar el nombre del autor original del software en el trabajo derivado e incluir una copia del texto de la licencia en cualquier redistribución del software.

Características de las licencias de software permisivas más importantes

Esta sección explica las diferencias entre la licencia MIT/X11, las licencias BSD más comunes y la licencia Apache 2.0, todas las cuales se utilizan ampliamente en la actualidad.

Licencia MIT/X11

La licencia MIT/X11 (text: https://opensource.org/license/mit), también conocida como licencia X11, es una de las licencias académicas mencionadas anteriormente. La licencia toma su nombre del software X Window System desarrollado por el Instituto de Tecnología de Massachusetts en 1987. Esta licencia es una de las licencias permisivas de software más antiguas y populares, en parte debido a su lenguaje simple y claro.

Esta licencia otorga al licenciatario los derechos a:

- Usar, modificar y distribuir el software
- Comercializar el software, con o sin modificaciones
- Publicar trabajos derivados bajo una licencia diferente, incluso como software de código cerrado

La única obligación del licenciatario es incluir el aviso de copyright y el texto de la licencia en el código fuente del software o de sus trabajos derivados.

La licencia MIT/X11 se considera compatible con todas las licencias copyleft más importantes, tanto débiles como fuertes. En particular, la licencia MIT/X11 es compatible con la

- Licencia pública general GNU (GPL), versiones 2.0 y 3.0
- Licencia pública general reducida (LGPL) GNU, versiones 2.0 y 3.0
- Licencia pública Mozilla (MPL)

Esto significa que los trabajos derivados de software originalmente licenciados bajo la licencia MIT/X11 pueden redistribuirse bajo una de las licencias mencionadas anteriormente o incluirse en proyectos publicados como los anteriores.

Licencia BSD de 2 cláusulas

La Licencia BSD de 2 Cláusulas (text: https://opensource.org/license/bsd-2-clause) deriva de la licencia BSD original, creada en 1980 por la Universidad de Berkeley en California como licencia para BSD, su Unix. -como sistema operativo.

La licencia destaca por su sencillez y, como su nombre indica, consta de sólo dos cláusulas breves. Es sustancialmente idéntica a la licencia MIT/X11. Al igual que la licencia MIT/X11, ésta requiere atribución en el software. Además, la licencia BSD de 2 cláusulas requiere que el licenciatario incluya el texto de la licencia en la documentación y otros recursos proporcionados al redistribuir el código.

En resumen, la licencia BSD de 2 cláusulas otorga los derechos a:

- Usar, modificar y distribuir el software
- Comercializar el software, con o sin modificación
- Publicar trabajos derivados bajo una licencia diferente, incluso como software de código cerrado

El licenciatario tiene la obligación de:

- Incluir el aviso de copyright, el texto de la licencia en el código fuente y binario del software o de sus trabajos derivados
- Incluir el aviso de copyright, el texto de la licencia en la documentación u otros materiales proporcionados con el software

La licencia BSD de 2 cláusulas se considera compatible con todas las principales licencias copyleft, tanto débiles como fuertes. En particular, la licencia BSD de 2 Cláusulas es compatible con:

- Licencia pública general GNU (GPL), versiones 2.0 y 3.0
- Licencia pública general reducida (LGPL) GNU, versiones 2.0 y 3.0
- Licencia pública Mozilla (MPL)

Por lo tanto, los trabajos derivados de software originalmente licenciados bajo la licencia BSD de 2 cláusulas pueden redistribuirse bajo una de las licencias mencionadas anteriormente o incluirse en proyectos publicados como los anteriores

Licencia BSD de 3 cláusulas

La licencia BSD de 3 cláusulas (text: https://opensource.org/license/bsd-3-clause) es otra variante más de la licencia BSD original y consta de solo tres cláusulas. La característica principal que distingue esta licencia de la licencia hermana BSD de 2 cláusulas es la “cláusula de no respaldo”, que tiene como objetivo evitar que el nombre del autor original sea explotado para distribuir o vender obras derivadas.

La licencia BSD de 3 cláusulas otorga al licenciatario los derechos a:

- Usar, modificar y distribuir el software
- Comercializar el software, con o sin modificaciones
- Publicar trabajos derivados bajo una licencia diferente, incluso como software de código cerrado

El licenciatario tiene la obligación de:

- Incluir el aviso de copyright y el texto de la licencia en el código fuente y binario del software o de sus trabajos derivados
- Incluir el aviso de copyright y el texto de la licencia en la documentación u otros materiales proporcionados con el software

- Abstenerse de citar el nombre del copyright titular o contribuyentes para respaldar o promocionar productos derivados de este software, sin permiso previo por escrito específico

Para explicar esta última obligación -la “cláusula de no respaldo”- pensemos en un escenario en el que una empresa o un individuo crea un trabajo derivado a partir de un software licenciado bajo una licencia de software permisiva. En ausencia de esta cláusula, el licenciatario podría promocionar la nueva obra indicando que se trata de un derivado de un software de un desarrollador conocido, con el fin de beneficiarse de su prestigio.

Al igual que su licencia hermana, la licencia BSD de 3 cláusulas se considera compatible con todas las licencias copyleft más importantes, tanto débiles como fuertes. En particular, la licencia BSD de 3 Cláusulas es compatible con:

- Licencia pública general GNU (GPL), versión 2.0 y 3.0
- Licencia pública general reducida (LGPL) GNU, versión 2 y 3
- Licencia pública Mozilla (MPL)

Por lo tanto, los trabajos derivados de software originalmente licenciados bajo la licencia BSD de 3 cláusulas pueden redistribuirse bajo una de las licencias mencionadas anteriormente o incluirse en proyectos publicados como los anteriores

Licencia Apache 2.0

La primera licencia de Apache fue desarrollada por Apache Software Foundation, una organización sin fines de lucro establecida en 1999, para distribuir su software y facilitar su inclusión en otros proyectos. La misión era garantizar el desarrollo colaborativo de software, adoptando la filosofía de código abierto con una perspectiva amigable para los negocios.

La licencia Apache 2.0 (text: https://www.apache.org/licenses/LICENSE-2.0), probablemente la más extendida entre las licencias de software permisivo, se diferencia de las licencias comentadas anteriormente en algunos aspectos relevantes. El aspecto principal se refiere a la atribución de derechos de patente a cada licenciatario, incluida una cláusula relacionada con la terminación de la licencia de patente, a fin de limitar y prevenir cualquier reclamación por daños y perjuicios por infracción de patente.

Otras dos obligaciones son importantes. El licenciatario debe liberar, bajo la misma licencia Apache 2.0, cualquier parte o componente del software que no haya sido modificado por el licenciatario. Además, el licenciatario debe colocar avisos destacados en cualquier archivo modificado que indiquen que cambió esos archivos.

La Free Software Foundation, que promueve licencias restrictivas, recomienda Apache 2.0 como

la mejor de las licencias de software permisivas para distribuir pequeñas cantidades de software y bibliotecas.

La licencia Apache 2.0 otorga al licenciatario:

- El derecho a usar, modificar y distribuir el software
- El derecho a comercializar el software, con o sin modificaciones
- El derecho a publicar trabajos derivados bajo una licencia diferente, incluso como software de código cerrado
- Una licencia de patente para usar, vender, importar y transferir de otro modo el software cubierto por una patente
- Una licencia de patente para usar, vender, importar y transferir de otro modo el software que de otro modo podría infringir la reivindicación de patente de un contribuyente

El licenciatario tiene la obligación de:

- Coloque avisos destacados en cualquier archivo modificado que indique que los archivos han sido modificados
- Libere todas las partes no modificadas del software original bajo la licencia Apache 2.0
- Incluya el aviso de derechos de autor y el texto de la licencia en el código fuente del software y sus trabajos derivados
- Incluir el aviso de derechos de autor y el texto de la licencia en la documentación y otros materiales proporcionados con el software.

La licencia Apache 2.0 se considera compatible sólo con algunas de las licencias copyleft más importantes. En particular, la licencia Apache 2.0 es compatible con:

- Licencia pública general GNU (GPL), versión 3.0 pero no 2.0
- Licencia pública general reducida (LGPL) GNU, versión 3.0 pero no la 2.0
- Licencia pública Mozilla (MPL), versión 2.0 pero no la versión 1.1

Esto significa que los trabajos derivados de software originalmente licenciados bajo la licencia Apache 2.0 pueden redistribuirse bajo una de las licencias compatibles mencionadas anteriormente o incluirse en proyectos lanzados bajo una de las licencias compatibles mencionadas anteriormente.

La compatibilidad limitada entre la licencia Apache 2.0 y otras licencias de código abierto se debe principalmente a la presencia de cláusulas relacionadas con la concesión de una licencia de patente al licenciatario.

Licencias de software permisivas en relación con otras licencias de código abierto

Ahora que tenemos una visión general de las características comunes y esenciales de las licencias de software permisivas, podemos pasar a examinar en qué se diferencian del software de dominio público y de las licencias copyleft.

Comparación con el software de dominio público

La primera distinción entre licencias de software permisivas y lanzamientos de software de dominio público radica en su propia existencia. El dominio público no es una licencia real, sino simplemente una forma de publicar software. En otras palabras, las obras de dominio público no tienen licencia.

La siguiente distinción radica en las obligaciones que las licencias permisivas imponen al licenciatario. De hecho, el usuario de una publicación de dominio público no tiene obligación alguna. Sin embargo, cualquiera que use, modifique o redistribuya software bajo una licencia de software permisiva debe cumplir con la obligación de atribución explicada anteriormente: el requisito de incluir el nombre del autor original y una copia del texto de la licencia en el software.

La consecuencia de la obligación de atribución es que ningún trabajo derivado que se origine a partir de software bajo una licencia de software permisiva puede ser liberado como dominio público, porque esto violaría (o al menos eludiría) el derecho del autor original a recibir atribución.

Comparación con Copyleft

La distinción entre licencias de software permisivas y licencias copyleft restrictivas es más compleja, especialmente considerando las diferencias entre las distintas licencias copyleft. Como se vio en lecciones anteriores, una diferencia importante separa las licencias copyleft *fuertes* (incluida la licencia pública general GNU (GPL), versiones 2.0 y 3.0) de las licencias copyleft *débiles* (incluida la licencia pública general reducida (LGPL) GNU, versiones 2.0 y 3.0, y la Licencia pública de Mozilla).

La principal diferencia entre licencias restrictivas y licencias permisivas radica en el principio copyleft, que es fundamental para las licencias restrictivas. De conformidad con este principio, el licenciatario que modifica software bajo licencia copyleft, y luego lo distribuye, debe necesariamente liberar, total o parcialmente, la obra derivada con la misma licencia que el software original. El incumplimiento de esta obligación da lugar a la infracción de los derechos de autor, con las consiguientes consecuencias legales.

Por el contrario, las licencias de software permisivas no imponen tal obligación. Quienes utilizan o planean lanzar software bajo este tipo de licencias pueden decidir libremente la licencia para su trabajo derivado, incluidas las licencias propietarias.

Copyleft fuerte

La diferencia entre las licencias de software permisivas y las licencias copyleft fuertes es más marcada que con las licencias copyleft débiles.

La diferencia esencial es que las licencias copyleft fuertes requieren que los trabajos derivados se publiquen bajo la misma licencia que el software original. Esto también se aplica si el software con licencia copyleft se integra en otro proyecto: Todo el trabajo derivado debe publicarse bajo la misma licencia copyleft sólida.

Copyleft débil

A diferencia de las licencias copyleft fuertes, las débiles sólo tienen un requisito parcial de publicar una obra derivada bajo la misma licencia. Para ser más específico, el licenciatario que redistribuye software publicado bajo una licencia copyleft débil debe aplicar la misma licencia a la parte de su trabajo derivada del original. Por ejemplo, una biblioteca basada en una biblioteca con licencia LGPL también debe tener licencia LGPL.

Las licencias copyleft débiles fueron concebidas precisamente para acercar sus características a las de las licencias de software permisivas. Sin embargo, se distinguen de las licencias de software permisivas en que estas últimas no imponen, en ningún caso, la obligación de mantener la misma licencia sobre una obra derivada o integrada.

Ejercicios guiados

1. ¿Cuáles son las dos obligaciones que generalmente imponen las licencias de software permisivas?

2. ¿Cuál de las siguientes licencias *no* es una licencia de software permisiva?

Licencia Apache 2.0	
Licencia LGPL	
Licencia MIT/X11	
BSD de 3 cláusulas	

3. ¿Qué distingue una licencia de software permisiva de una licencia copyleft?

El software lanzado bajo una licencia copyleft no se puede distribuir, mientras que el software bajo una licencia de software permisiva sí se puede distribuir.	
Las obras derivadas de software bajo una licencia copyleft no pueden publicarse bajo una licencia propietaria, mientras que el software bajo una licencia de software permisiva sí puede.	
Las licencias copyleft están legalmente reconocidas sólo en los Estados Unidos de América, mientras que las licencias de software permisivas están reconocidas globalmente.	

4. ¿Qué licencia de software permisiva otorga una licencia de patente para usar, vender, importar y transferir de otro modo el software cubierto por una patente?

licencia apache 2.0
Licencia BSD de 2 cláusulas
Licencia MIT/X11
Licencia BSD de 3 cláusulas

5. Cuando el software se lanza bajo la licencia MIT/X11, ¿se puede distribuir bajo una licencia propietaria y vender un trabajo derivado basado en el software?

Ejercicios exploratorios

1. Está modificando el software distribuido bajo la licencia Apache 2.0 y redistribuyendo el trabajo derivado bajo una licencia propietaria. ¿Qué pasos debe seguir para cumplir con las obligaciones de la licencia Apache 2.0?

2. Nombra al menos tres ejemplos de proyectos populares publicados bajo licencias de software permisivas.

3. ¿Por qué no se puede distribuir, bajo una licencia LGPL 2.0, software que incluya componentes que se lanzaron originalmente bajo la licencia MIT/X11, la licencia Apache 2.0 y la licencia BSD de 2 cláusulas? ¿Qué diferente licencia copyleft débil se puede utilizar para lanzar el software?

Resumen

En esta lección has aprendido: * Qué son las licencias de software permisivo, los derechos que otorgan y las obligaciones que proporcionan * Las diferencias entre las licencias de software permisivo y otras licencias de código abierto * Características de las licencias de software permisivo más populares * Compatibilidad del software permisivo licencias con otras licencias de código abierto

Respuestas a ejercicios guiados

1. ¿Cuáles son las dos obligaciones que generalmente imponen las licencias de software permisivas?

   La obligación de indicar el nombre del autor original del software y de incluir una copia del texto de la licencia en la obra derivada.

2. ¿Cuál de las siguientes licencias *no* es una licencia de software permisiva?

Licencia Apache 2.0	
Licencia LGPL	X
Licencia MIT/X11	
BSD de 3 cláusulas	

3. ¿Qué distingue una licencia de software permisiva de una licencia copyleft?

El software lanzado bajo una licencia copyleft no se puede distribuir, mientras que el software bajo una licencia de software permisiva sí se puede distribuir.	
Las obras derivadas de software bajo una licencia copyleft no pueden publicarse bajo una licencia propietaria, mientras que el software bajo una licencia de software permisiva sí puede.	X
Las licencias copyleft están legalmente reconocidas sólo en los Estados Unidos de América, mientras que las licencias de software permisivas están reconocidas globalmente.	

4. ¿Qué licencia de software permisiva otorga una licencia de patente para usar, vender, importar y transferir de otro modo el software cubierto por una patente?

Licencia Apache 2.0	X
Licencia BSD de 2 cláusulas	
Licencia MIT/X11	

Licencia BSD de 3 cláusulas	

5. Cuando el software se lanza bajo la licencia MIT/X11, ¿se puede distribuir bajo una licencia propietaria y vender un trabajo derivado basado en el software?

   Sí.

Respuestas a ejercicios exploratorios

1. Está modificando el software distribuido bajo la licencia Apache 2.0 y redistribuyendo el trabajo derivado bajo una licencia propietaria. ¿Qué pasos debe seguir para cumplir con las obligaciones de la licencia Apache 2.0?

   Debería:

   - Insertar en cada archivo modificado un aviso que acredite que los archivos han sido modificados.
   - Liberar todas las partes no modificadas del software original bajo la licencia Apache 2.0.
   - Incluir el aviso de copyright y el texto de la licencia en el código fuente del software o de sus trabajos derivados.
   - Incluya el aviso de derechos de autor y el texto de la licencia en la documentación y otros materiales proporcionados con la distribución del software.

2. Nombra al menos tres ejemplos de proyectos populares publicados bajo licencias de software permisivas.
   - Marco web angular: licencia MIT/X11
   - Ruby on Rails - Licencia MIT/X11
   - Servidor HTTP Apache - Licencia Apache 2.0
   - Kubernetes: licencia Apache 2.0

3. ¿Por qué no se puede distribuir, bajo una licencia LGPL 2.0, software que incluya componentes que se lanzaron originalmente bajo la licencia MIT/X11, la licencia Apache 2.0 y la licencia BSD de 2 cláusulas? ¿Qué diferente licencia copyleft débil se puede utilizar para lanzar el software?

   Aunque la licencia MIT/X11 y la licencia BSD de 2 cláusulas son compatibles con la licencia LGPL 2.0, la licencia Apache 2.0 no lo es. El software que incluye componentes publicados bajo la licencia MIT/X11, la licencia Apache 2.0 y la licencia BSD de 2 cláusulas se puede publicar bajo la licencia LGPL 3.0, porque es compatible con todas esas licencias de software permisivas.

Linux Professional Institute

Tema 053: Licencias de contenido abierto

053.1 Conceptos de licencias de contenido abierto

Referencia al objetivo del LPIs

Open Source Essentials version 1.0, Exam 050, Objective 053.1

Peso

2

Áreas de conocimiento clave

- Comprender los tipos de contenido abierto
- Comprender qué constituye contenido sujeto a derechos de autor
- Comprender las obras derivadas de materiales protegidos por derechos de autor
- Comprender la necesidad de licencias de contenido abierto
- Conocimiento de las marcas

Lista parcial de archivos, términos y utilidades

- Documentación
- Imágenes
- Obra de arte
- Mapas
- Música
- Vídeos
- Diseños y especificaciones de hardware
- Bases de datos
- Flujos de datos

- Fuentes de datos

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	053 Licencias de contenido abierto
Objectivo:	053.1 Conceptos de Licencias de Contenido Abierto
Lección:	1 de 1

Introducción

Piense en esta situación hipotética: Frank es un escritor que quiere que algunas de sus historias estén disponibles para cualquiera en Internet, pero bajo ciertas condiciones. Quiere que su trabajo sea libre de costo. Quiere que cualquiera pueda utilizar las historias sin pedir permiso, pero no quiere que su trabajo se utilice comercialmente. Finalmente, desea que las personas le de el crédito si las historias se utilizan para algún propósito. Le gustaría hacerlo sin involucrarse en complicados procesos legales. Más adelante veremos por qué un escritor podría querer hacer estas cosas.

Emma es una cineasta que enseña a algunos estudiantes cómo producir cortometrajes. Entre otras cosas, los alumnos necesitan una historia para crear sus obras. Emma conoce las historias de Frank y cree que son perfectas para su clase.

¿Qué puede hacer Frank para permitirle a Emma usar sus historias? Esta situación y muchas más pueden resolverse utilizando un modelo de *licencia de contenido abierto*. Hoy en día, cualquiera puede reutilizar millones de obras protegidas por derechos de autor sin el consentimiento explícito del titular de los derechos o el pago de una tarifa de licencia, gracias a su distribución

bajo licencias de contenido abierto. Recursos como Wikipedia, Flickr, OpenStreetMap, Unsplash y Jamendo son algunos ejemplos de las muchas plataformas que utilizan este modelo de licencia.

Según una definición muy amplia, *contenido abierto* se refiere a cualquier trabajo (incluidos, por ejemplo, películas, música, imágenes, textos, bases de datos, conjuntos de datos, documentación, mapas y diseños de hardware) cuyo libre uso y redistribución está permitido según la ley de derechos de autor. Esto también incluye las obras originalmente protegidas cuyo plazo de protección ha expirado y que, por tanto, son de dominio público. Una definición algo más estrecha presupone que una obra ha sido colocada explícitamente bajo una licencia de contenido abierto por parte de su autor, lo que permite el uso y distribución de la obra sin ningún pago o permiso. Por lo tanto, el modelo de contenido abierto no se opone a la ley de derechos de autor, sino que la complementa.

Fundamentos de derechos de autor

Debido a que las licencias de contenido abierto se basan en los derechos de autor, es necesario aprender algunos aspectos de la ley de derechos de autor para comprender por qué se necesitan las licencias y qué permiten.

Los derechos de autor son parte de una categoría legal conocida como *propiedad intelectual*. Los derechos de autor protegen las obras originales al otorgar a sus creadores derechos legales exclusivos para controlar ciertos usos de sus obras por parte de otras personas y corporaciones. Generalmente, esto significa que nadie más puede copiar, distribuir, representar públicamente, adaptar o hacer casi cualquier otra cosa que no sea ver o leer la obra sin el permiso del titular de los derechos de autor.

Los fundamentos que examinaremos en esta sección incluyen qué está protegido por derechos de autor, junto con quién controla los derechos y puede otorgar permiso para reutilizar una obra protegida por derechos de autor como Emma quiere hacer en nuestro ejemplo.

Como lo afirma explícitamente la Organización Mundial de la Propiedad Intelectual:

> Los derechos de autor protegen dos tipos de derechos. Los derechos económicos permiten a los titulares de derechos obtener una compensación financiera por el uso de sus obras por parte de terceros. Los derechos morales permiten a los autores y creadores tomar determinadas acciones para preservar y proteger su vínculo con su obra. El autor o creador puede ser titular de los derechos patrimoniales o esos derechos pueden transferirse a uno o más titulares de derechos de autor. Muchos países no permiten la transferencia de derechos morales. Los derechos de autor sólo protegen la expresión de hechos o ideas. No permite que el titular de los derechos de autor sea propietario o controle la idea exclusivamente. Por ejemplo, una ilustración puede tener derechos de autor pero no la idea que la originó.

> — Organización Mundial de la Propiedad Intelectual, Comprensión de los derechos de autor y derechos conexos

Los derechos económicos protegidos por derechos de autor duran mucho tiempo, generalmente décadas después de la muerte del creador. Los derechos morales nunca caducan.

Los derechos de autor otorgan derechos sobre obras creativas, como creaciones literarias y artísticas, que deben cumplir con un cierto estándar de originalidad. La obra debe ser creación de su creador y no copiada de otra obra. (Existen debates en curso sobre cuánta inteligencia artificial se puede incluir en una obra y aún así considerarla original).

Los derechos de autor protegen únicamente la expresión de hechos o ideas. No permite que el titular de los derechos de autor sea propietario o controle la idea exclusivamente. Por ejemplo, una ilustración puede tener derechos de autor, pero no la idea que la originó.

Los derechos de autor son automáticos cuando una obra se crea y fija en alguna forma tangible, por ejemplo, una obra de arte digital o una canción. Esto significa que los derechos se otorgan al creador sin siquiera registrar formalmente la obra.

Las personas que promueven las licencias de contenido abierto quieren promover una cultura libre y el desarrollo de un bien común digital. Han encontrado que el sistema de derechos de autor es demasiado restrictivo y rígido tanto para los usuarios como para los creadores. Al crear licencias estándar fáciles de usar, los defensores del contenido abierto simplifican el uso y la distribución de obras protegidas por derechos de autor.

¿Qué puede tener derechos de autor?

Las leyes de derechos de autor varían de un país a otro. Sin embargo, existen acuerdos internacionales para estandarizar las leyes de derechos de autor. Las obras literarias y artísticas originales pueden tener derechos de autor: por ejemplo, obras de arte, música, fotografía, cine, televisión, literatura y programación. Las reglas particulares para decidir qué está protegido por derechos de autor y qué tan original es una obra varían según la región.

En ocasiones, estas categorías pueden ser muy generales y aplicarse a obras que tienen elementos tanto creativos como estrictamente funcionales: por ejemplo, un cortometraje se considera obra artística. Sin embargo, algunas partes del mismo pueden no tener derechos de autor si no cumplen con el estándar de originalidad.

Obras Derivadas

Sigamos con la situación hipotética del comienzo de la lección: Frank finalmente decidió publicar sus historias bajo una licencia de contenido abierto para que puedan usarse en los términos que eligió. Como las historias estaban bajo esa licencia, Emma las usó en su clase y decidió mostrar

algunos de los cortometrajes de sus alumnos durante un festival de cine. En este caso, los contenidos creados por los estudiantes pueden considerarse *trabajos derivados*.

Una obra derivada o *adaptación* es una obra basada en obras existentes, como una traducción, arreglo musical, dramatización, ficcionalización, versión cinematográfica, grabación de sonido, reproducción de arte o cualquier otra forma en la que se transforme o adapte la obra original. La obra derivada se convierte en una segunda obra separada, independiente en forma de la primera. La transformación, modificación o adaptación de la obra debe ser sustancial para ser original y así estar protegida por derechos de autor.

Funciones comunes de licencias de contenido abierto

Cada licencia de contenido abierto afirma los derechos de autor del creador y asegura que sin una licencia del autor, cualquier persona que utilice la obra estaría violando los derechos de autor. Por lo tanto, dichas licencias funcionan dentro del sistema global de derechos de autor en lugar de intentar anularlo.

Las licencias de contenido abierto también garantizan que se otorguen los créditos adecuados al autor del trabajo. Si los destinatarios del trabajo lo distribuyen a un tercero, deben asegurarse de que se reconozca y acredite al autor original. Además, cuando los destinatarios modifiquen la obra, la obra derivada debe mencionar claramente al autor del original y mencionar dónde se puede encontrar el original.

A diferencia de la mayoría de las licencias de derechos de autor, que imponen condiciones restrictivas sobre los usos de la obra, las licencias de contenido abierto permiten a los usuarios tener ciertas libertades al otorgarles derechos. Algunos de estos derechos son comunes a prácticamente todas las licencias de contenido abierto, como el derecho a copiar la obra y el derecho a distribuirla. Dependiendo de la licencia, el usuario también puede tener derecho a modificar la obra, crear obras derivadas, realizar la obra, exhibir la obra y distribuir las obras derivadas.

Las licencias de contenido abierto pueden controlar las obras derivadas. Estas licencias normalmente incluyen el derecho a crear una obra derivada y distribuirla en cualquier medio. Si una persona licencia una pintura bajo una licencia de contenido abierto, también se le puede otorgar el derecho de basar otra imagen en ella. Estos permisos se otorgan con la condición de que otros puedan usar el trabajo derivado libremente, al igual que el trabajo original.

Por lo tanto, una licencia de contenido abierto normalmente garantiza que las obras derivadas sean licenciadas bajo los términos y condiciones de la misma licencia de contenido abierto. Pero esta obligación no es aplicable cuando la obra está incluida en una recopilación. Por ejemplo, si una persona hace un álbum de canciones, una de las cuales tiene una licencia de contenido

abierto, no todas las canciones tienen que tener la licencia bajo los mismos términos.

Otro aspecto importante de las licencias de contenido abierto es el control del uso comercial de una obra. Las personas pueden licenciar sus obras bajo una licencia de contenido abierto y al mismo tiempo restringir los derechos a fines no comerciales. Alternativamente, las personas también pueden otorgar todos los derechos, incluido el derecho a utilizar la obra comercialmente.

Las licencias de contenido abierto no impiden que las personas ganen dinero con su trabajo. Si una obra está bajo una licencia no comercial, un editor u otra entidad comercial puede publicar la obra mediante un acuerdo con los titulares de los derechos de autor antes de hacerlo. En otras palabras, incluso después de publicar una obra sin fines comerciales, los autores pueden vender los derechos de autor a una entidad con fines de lucro, siempre que no se excluya el uso continuo y no comercial.

Importancia de las licencias de contenido abierto

¿Cuáles son algunos de los beneficios de un modelo de contenido abierto y por qué la gente utilizaría una licencia de contenido abierto para distribuir su trabajo creativo en lugar de depender del modelo tradicional de derechos de autor?

Las licencias de contenido abierto permiten que las obras circulen más ampliamente que si estuvieran restringidas de forma predeterminada. Por lo tanto, nuevos artistas podrían beneficiarse de estas licencias, ya que pueden volverse más populares y reconocidos por más personas. Al renunciar a ciertos controles (y posiblemente a los ingresos que los acompañan), los creadores de contenido pueden ser invitados a más programas, obtener más patrocinios, realizar más colaboraciones, etc.

Las licencias de contenido abierto pueden resultar prácticas en la era de Internet, porque los creadores pueden publicar sus obras sin depender de otra persona o institución. Por ejemplo, un fotógrafo que quiera exponer sus fotografías podría publicarlas en un sitio web personal. De esta manera, pueden establecerse y ser conocidos por un público más amplio sin necesidad de contratar una agencia o galería para hacerlo.

Al elegir la licencia adecuada, los titulares de derechos pueden maximizar la distribución y mantener el control de la comercialización de sus obras. Si las personas quieren utilizar una obra comercialmente, el creador del contenido puede conservar el derecho de otorgar o negar el permiso. Incluso si a otros se les prohíbe el uso comercial, los titulares de derechos aún pueden usar su trabajo comercialmente.

Además de la posibilidad de una distribución mucho más amplia de una obra, las licencias de contenido abierto también aumentan la seguridad jurídica para los usuarios y reducen

significativamente los costos de transacción legal.

Las personas también pueden utilizar modelos de financiación que no dependan del uso de una licencia no comercial. Por ejemplo, muchos artistas y creadores utilizan el crowdfunding para financiar su trabajo antes de publicarlo bajo una licencia permisiva. Otros utilizan un modelo en el que el contenido básico es gratuito, pero los extras, como las versiones impresas o el acceso especial a un sitio web exclusivo para miembros, son sólo para clientes que pagan.

Las licencias de contenido abierto permiten a las personas distribuir sus obras a cualquier persona y en cualquier medio y formato, como sitios web, fotocopias, CD o libros, sin restricciones.

Las licencias de contenido abierto establecen automáticamente una licencia entre autores y usuarios. Sin una licencia de contenido abierto, compartir obras a través de otra fuente en línea requeriría un acuerdo contractual individual entre autores y usuarios.

Marcas comerciales y derechos de autor

Las marcas comerciales, al igual que los derechos de autor, son un tipo de propiedad intelectual. Pero la ley que protege una marca es diferente de la que protege los derechos de autor. Las marcas comerciales protegen las marcas, palabras relacionadas como nombres de marcas, logotipos, símbolos e incluso sonidos y colores que se utilizan para distinguir determinados bienes y servicios de otros. Una marca puede ser cualquier elemento especial utilizado para promocionar y distinguir los negocios y los servicios o productos vendidos de otros.

El titular de una marca generalmente puede impedir que otros utilicen estos elementos de marca si el público se siente confundido. La ley de marcas ayuda a los productores de bienes y servicios a proteger su reputación y protege al público brindándoles una forma sencilla de diferenciar entre productos y servicios similares.

Las marcas pueden registrarse oficialmente o protegerse automáticamente según el derecho consuetudinario. Esto significa que una marca existe tan pronto como se utiliza, pero una marca de derecho consuetudinario no ofrece la misma protección legal que una marca registrada. Por eso muchas empresas registran sus marcas.

Los derechos de autor y las marcas pueden coexistir. Por ejemplo, Wikipedia es una marca registrada y su logotipo también está protegido por la ley de derechos de autor como obra de arte o creación original.

Ejercicios guiados

1. ¿Se pueden utilizar licencias de contenido abierto para evitar los derechos de autor?

2. ¿Qué se considera contenido abierto?

3. ¿Qué es una obra derivada?

Ejercicios exploratorios

1. ¿Qué harías si quisieras publicar una obra y permitir que las personas la utilicen para cualquier propósito siempre que la redistribuyan bajo los mismos derechos y condiciones?

2. ¿Se pueden distribuir trabajos derivados comerciales basados en trabajos publicados bajo una licencia de contenido abierto?

Resumen

En esta lección, aprendiste:

- Fundamentos de la ley de derechos de autor
- Qué se puede considerar contenido protegido por derechos de autor
- Qué es una obra derivada o adaptación
- Características comunes que comparten las licencias de contenido abierto
- Tipos de contenido abierto
- Importancia y beneficios del modelo de licencia de contenido abierto
- Copyright y marcas registradas como propiedad intelectual tipos

Respuestas a ejercicios guiados

1. ¿Se pueden utilizar licencias de contenido abierto para evitar los derechos de autor?

   Las licencias de contenido abierto no se pueden utilizar para evitar los derechos de autor. Ellas son un tipo de licencia de derechos de autor que otorga algunos derechos bajo ciertas condiciones, manteniendo los derechos exclusivos del titular de los derechos de autor.

2. ¿Qué se considera contenido abierto?

   El contenido abierto puede ser cualquier trabajo protegido por derechos de autor publicado bajo una licencia de contenido abierto. Este contenido puede ser películas, música, imágenes, textos, bases de datos, documentación, mapas y diseños de hardware, y otras creaciones. Estos tipos de licencias otorgan derechos como uso, redistribución, realización de trabajos derivados y uso comercial o no comercial sin ningún permiso especial.

3. ¿Qué es una obra derivada? + Una obra derivada, también llamada adaptación, es una obra basada en una obra existente en la que se transforma o adapta la obra original. Las traducciones, los arreglos musicales, las dramatizaciones, las versiones cinematográficas, las grabaciones de sonido y las reproducciones de arte son ejemplos de obras derivadas.

Respuestas a ejercicios exploratorios

1. ¿Qué harías si quisieras publicar una obra y permitir que las personas la utilicen para cualquier propósito siempre que la redistribuyan bajo los mismos derechos y condiciones?

   Puedes poner la obra a disposición bajo una licencia tipo copyleft. De esta manera, todas las obras derivadas del original también deben estar abiertas, lo que requiere que se publiquen bajo la misma licencia o cualquier otra compatible.

2. ¿Se pueden distribuir trabajos derivados comerciales basados en trabajos publicados bajo una licencia de contenido abierto?

   Depende de la licencia bajo la cual esté disponible la obra original. Si la licencia establece que puede utilizar obras derivadas para cualquier propósito, incluso comercialmente, entonces puede hacerlo.

053.2 Licencias Creative Commons

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 053.2

Peso

2

Áreas de conocimiento clave

- Comprender el concepto de licencias Creative Commons
- Comprender los tipos de licencia Creative Commons y sus combinaciones
- Comprender los derechos otorgados por las licencias Creative Commons
- Comprender las obligaciones creadas por las licencias Creative Commonss

Lista parcial de archivos, términos y utilidades

- Dedicación de Dominio Público (CC0)
- Atribución Creative Commons (CC BY)
- Creative Commons Atribución-CompartirIgual (CC BY-SA)
- Reconocimiento-No comercial de Creative Commons (CC BY-NC)
- Creative Commons Atribución-No Comercial-CompartirIgual (CC BY-NC-SA)
- Atribución Creative Commons-SinDerivadas (CC BY-ND)
- Creative Commons Atribución-NoComercial-SinDerivadas (CC BY-NC-ND)

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	053 Licencias de contenido abierto
Objectivo:	053.2 Licencias Creative Commons
Lección:	1 de 1

Introducción

El éxito del software libre desde los años 1990, junto con el triunfo simultáneo de Internet, han despertado en otros sectores el deseo de condiciones similares para apoyar el desarrollo y la distribución de obras creativas, es decir, obras que están fundamentalmente sujetas a derechos de autor. Estos deseos reflejan una variedad de intereses.

Los artistas creativos quieren determinar por sí mismos las condiciones bajo las cuales otros pueden utilizar sus obras. Al mismo tiempo, sin embargo, también tienen interés en utilizar el trabajo de otros para sí mismos, como siempre lo han hecho los procesos creativos, por ejemplo citando, editando o adaptando. Para los destinatarios de las obras surgen dudas sobre las posibilidades de uso, por ejemplo, si una obra puede reproducirse o transmitirse y de qué forma.

Además de aclarar estas cuestiones prácticas, las regulaciones deben formularse de tal manera que estén en consonancia con las normas legales aplicables -principalmente la ley de derechos de autor-, lo que se hace más difícil por las diversas formas en que se regula internacionalmente la ley de derechos de autor.

Por último, pero no menos importante, las normas deben ser fáciles de entender y sencillas de

aplicar, con el fin de acelerar los procesos creativos y dar seguridad jurídica a los creadores y destinatarios.

Origen y objetivos de Creative Commons

Las primeras licencias en el ámbito del software libre, sobre todo la licencia pública general GNU, fueron pioneras, ya que lograron proporcionar un marco jurídico fiable para procesos completamente nuevos relacionados con el desarrollo colaborativo de software.

En 2001, un grupo liderado por el profesor de derecho Lawrence Lessig de la Facultad de Derecho de Stanford fundó una organización sin fines de lucro llamada *Creative Commons* (CC), inspirada en el software libre. El objetivo de la organización se resume en el sitio web del proyecto de la siguiente manera:

> Creative Commons es una organización global sin fines de lucro que permite compartir y reutilizar la creatividad y el conocimiento mediante el suministro de herramientas legales gratuitas.
>
> — Creative Commons, sitio web (Preguntas frecuentes)

Con el término “bienes comunes”, que se refiere a una forma de actividad económica comunitaria ya documentada en la Edad Media, la organización enfatiza una necesidad históricamente verificable y profundamente humana de una acción colectiva orientada hacia el bien común. Por lo tanto, su tarea autoimpuesta es crear un marco legal moderno para el trabajo creativo. Creative Commons traslada las demandas y soluciones del movimiento del software libre a otros campos creativos como la literatura, las artes escénicas (pintura, gráfica, fotografía, vídeo, etc.) y la música, pero también la documentación y el trabajo científico -en términos generales, todas las obras que están sujetas a derechos de autor.

Al igual que la Free Software Foundation, Creative Commons también elige implementar sus objetivos a través de *licencias*: compilaciones de especificaciones estandarizadas y legalmente vinculantes que los creadores asignan explícitamente a sus obras, generalmente antes de “lanzarlas” al público, es decir, publicarlas.

El principio “todos los derechos reservados” consagrado en la legislación estadounidense sobre derechos de autor se considera demasiado restrictivo en vista de las crecientes posibilidades técnicas disponibles para los procesos creativos. Los propios creadores a menudo no tienen claro hasta qué punto se les permite aprovechar el trabajo de otros sin exceder los límites de la ley de derechos de autor y, en el peor de los casos, exponerse a ser procesados. Creative Commons contrasta este régimen de mano dura con el principio de “algunos derechos reservados”: los creadores nombran los derechos que se reservan y, por lo tanto, renuncian a todos los demás.

La combinación de sólo cuatro sencillas condiciones básicas (módulos) da como resultado un total de seis licencias entre las que los autores pueden seleccionar y asignar la adecuada a su trabajo.

Los módulos de licencia Creative Commons

Los cuatro módulos que acabamos de mencionar corresponden a cuatro decisiones básicas simples que toma un autor con respecto al uso y distribución de una obra. Cada módulo se puede representar mediante una abreviatura de dos letras y un símbolo. La definición respectiva es muy breve y comprensible incluso para los legos en el ámbito jurídico, lo que contribuye en gran medida a la popularidad de las licencias CC. Sin embargo, en casos individuales, las cláusulas pueden dar lugar a cuestiones no resueltas o zonas grises.

Atribución (BY)

> Debe dar el crédito apropiado, proporcionar un enlace a la licencia e indicar si se realizaron cambios. Puede hacerlo de cualquier manera razonable, pero no de ninguna manera que sugiera que el licenciante lo respalda a usted o su uso.
>
> — Creative Commons, Atribución

La palabra inglesa “by” indica que se debe nombrar a los autores, es decir, la obra debe atribuirse claramente a los autores si se utiliza, reproduce o transmite de cualquier forma. El módulo BY es el único obligatorio en *todas* las licencias CC. En otras palabras, no hay ningún trabajo bajo las seis licencias CC estándar que pueda escapar a la atribución.

Este simple requisito puede causar problemas prácticos en proyectos colaborativos desarrollados a través de Internet. Por ejemplo, en trabajos derivados de la enciclopedia en línea Wikipedia, es bastante cuestionable cómo se debe llevar a cabo “apropiadamente” la “atribución” con miles de contribuyentes.

No Comercial (NC)

> No puede utilizar el material con fines comerciales.
>
> — Creative Commons, No comercial

La intención del módulo *NonCommercial* parece clara a primera vista: pretende evitar que otros se apropien de una obra disponible gratuitamente con fines comerciales. A menudo se trata de trabajos que se han desarrollado con fondos públicos, por ejemplo en las universidades. Estos deben estar “protegidos” de la comercialización para garantizar su calidad y libre disponibilidad a largo plazo.

De hecho, el módulo NC a menudo genera incertidumbre en la práctica, ya que en casos individuales no está claro cuándo un uso es realmente comercial. Los profesores que utilizan materiales bajo una licencia CC con un módulo NC, por ejemplo, ya se encuentran en una zona legal gris si enseñan en una escuela privada o una universidad privada a la que los estudiantes tienen que pagar para asistir.

Muchos críticos del módulo NC también rechazan el argumento de que los materiales gratuitos pasan a ser "no libres" a través del uso comercial, porque las obras todavía están disponibles de forma gratuita.

No derivados (ND)

> Si remezcla, transforma o construye sobre el material, no puede distribuir el material modificado.
>
> — Creative Commons, NoDerivs

Derivatives (abreviado: *Derivs*), es decir, trabajos derivados, ya se han mencionado en otras lecciones. En el contexto de Creative Commons, el término también se refiere a la creación de nuevas obras sobre la base de obras existentes cubiertas por derechos de autor; Puede tratarse, por ejemplo, de materiales didácticos que un profesor adapta a sus clases o de la traducción de una novela a otro idioma. *NoDerivs* excluye categóricamente la distribución de modificaciones o adaptaciones de este tipo: La obra sólo podrá transmitirse en la forma publicada por el autor.

Compartir por igual (Share Alike - SA)

> Si remezcla, transforma o construye sobre el material, debe distribuir sus contribuciones bajo la misma licencia que el original.
>
> — Creative Commons, ShareAlike

El módulo *ShareAlike* indica que una obra podrá ser compartida únicamente bajo las mismas condiciones que las de la licencia asignada. SA se considera la contraparte del principio copyleft en las licencias de software libre, que garantiza que las libertades otorgadas por la licencia también se aplican sin cambios a las obras derivadas.

El módulo SA es particularmente interesante en combinación con otros módulos porque el requisito SA también actualiza otras condiciones definidas, como veremos a medida que analicemos las distintas licencias.

Las licencias básicas Creative Commons

Las combinaciones de los módulos recién presentados dan como resultado un total de seis licencias. Solo hay seis porque no todas las combinaciones posibles de los cuatro módulos tienen sentido: por ejemplo, los requisitos de compartir incluso obras modificadas en las mismas condiciones (ShareAlike) y la prohibición de adaptaciones (NoDerivs) son mutuamente excluyentes. En consecuencia, no existe ninguna licencia que contenga ambos módulos. Dado que la atribución del autor también es un componente de todas las licencias, el resultado son las llamadas *licencias principales* que analizaremos a continuación.

La lista de estas licencias también se puede imaginar como una escala de “muchas libertades” a “pocas libertades”, ya que un autor responde paso a paso a preguntas sobre las posibilidades de uso o sus restricciones en relación con la obra y, por tanto, finalmente recibe la licencia deseada.

Atribución Creative Commons (CC BY)

La atribución del autor ya se ha mencionado como un componente obligatorio de todas las licencias, por lo que la primera licencia, *CC Attribution*, incluye sólo este módulo. Por tanto, una obra queda liberada para cualquier forma de uso, modificación y distribución siempre que se cumpla el requisito de atribución.

Figure 5. CC BY Icon

Por ejemplo, si un autor ha publicado un poema bajo CC BY, un editor podría incluirlo en una colección de poemas sin consultar al autor y sin ninguna obligación financiera y distribuirlo a través de librerías, por ejemplo, siempre que este poema esté claramente marcado como obra del autor.

Creative Commons Attribution-ShareAlike (CC BY-SA)

CC Attribution-ShareAlike corresponde a CC BY, pero lo complementa con los requisitos de *ShareAlike*, es decir, la distribución de versiones modificadas de la obra en las mismas condiciones.

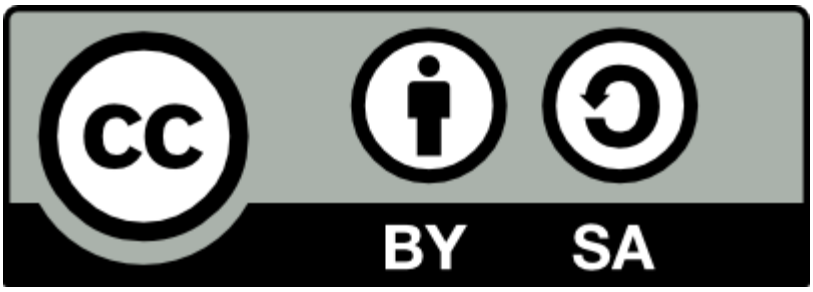

Figure 6. CC BY-SA Icon

Por ejemplo, si un cantante coloca su canción bajo la licencia CC BY-SA, otra banda puede versionar o adaptar este trabajo, siempre que publique a su vez su versión de la canción bajo CC BY-SA u otra licencia que sea compatible con él.

Creative Commons Attribution-NonCommercial (CC BY-NC)

La *CC Atribución-No Comercial* es la primera licencia de la serie que vincula el uso de una obra a una prohibición al excluir su distribución y adaptación con fines comerciales.

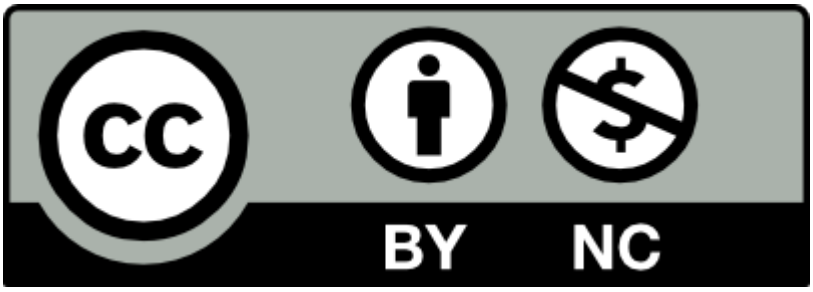

Figure 7. CC BY-NC Icon

En nuestro ejemplo, a una banda se le permitiría adaptar la canción del cantante, pero no utilizar esta adaptación comercialmente, por ejemplo, no venderla en sus propios discos ni tocarla en conciertos por los que reciba una tarifa. Sin embargo, a la banda se le permitiría prohibir adicionalmente las adaptaciones de su versión, es decir, publicar la adaptación bajo la licencia CC BY-NC-ND.

Creative Commnons Attribution-NonCommercial-ShareAlike (CC BY-NC-SA)

En el caso de *CC Attribution-NonCommercial-ShareAlike*, la prohibición de uso comercial está vinculada a la distribución en las mismas condiciones. Siguiendo con nuestro ejemplo: aunque la banda puede hacer un cover de la canción del cantante, no está permitido agregar ND a la versión porque el arreglo debe distribuirse en las mismas condiciones.

Figure 8. CC BY-NC-SA Icon

Creative Commons Attribution-NoDerivs (CC BY-ND)

Al igual que NonCommercial, *CC Attribution-NoDerivs* también se basa en una prohibición, en este caso, la prohibición de modificar o adaptar una obra. Por lo tanto, a la banda de nuestro ejemplo no se le permitiría distribuir una versión de la canción del cantante.

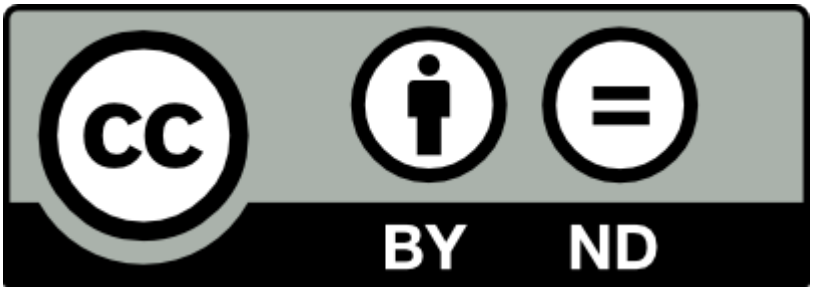

Figure 9. CC BY-ND Icon

El uso de licencias CC con cláusulas ND (es decir, la licencia CC BY-ND y la licencia CC BY-NC-ND que se presenta a continuación) se ve con escepticismo, particularmente en el campo científico, porque pueden obstaculizar el intercambio de conocimientos necesarios para el progreso científico. Incluso en el sitio web Creative Commons las licencias con un módulo ND se describen como incompatibles con el principio de acceso abierto tan popular en la ciencia. Como ejemplo de la tendencia demasiado restrictiva de la ND se cita que incluso las traducciones de artículos científicos se consideran trabajos derivados y, por tanto, están prohibidas.

Creative Commons Attribution-NonCommercial-NoDerivs (CC BY-NC-ND)

La más restrictiva de las licencias CC, *CC Attribution-NonCommercial-NoDerivs*, combina la prohibición del uso comercial con la de distribución de obras derivadas. En términos positivos, esto significa que una obra puede reproducirse y distribuirse únicamente en la forma publicada por el autor y puede utilizarse únicamente con fines no comerciales.

Figure 10. CC_BY-NC-ND_Icon

La cantante de nuestro ejemplo impide así que una banda haga un cover de su canción (ND), pero al mismo tiempo corre el riesgo de que su canción sea excluida de cualquier plataforma que genere ingresos de clientes o publicidad.

Creative Commons Zero (CC0) y la marca de dominio público

Ya se ha mencionado que la legislación sobre derechos de autor está regulada de manera muy diferente en distintos países y jurisdicciones. Según la legislación estadounidense, por ejemplo, un titular de derechos puede renunciar completamente a sus derechos de autor. Al hacerlo, transfieren su trabajo al llamado *dominio público*, es decir, a un uso general y sin restricciones.

En la mayoría de los países europeos, sin embargo, además de los meros derechos de uso, los derechos de autor también incluyen derechos personales, que generalmente no son transferibles y, por lo tanto, el autor no puede renunciar a ellos. Si un autor desea hacer pública una obra, renuncia a todos los derechos de uso, pero sigue siendo el autor.

Además, existen obras cuyo uso no está sujeto a ninguna restricción per se. Entre ellas se incluyen, obras cuyo plazo legal de protección ha expirado (como una novela 70 años después de la muerte del autor), u obras que nunca estuvieron protegidas en principio (como los textos legales).

Creative Commons proporciona dos de las llamadas *herramientas de dominio público* para identificar obras que están disponibles para el público en general sin restricciones.

Creative Commons Zero (CC0) no impone prácticamente ninguna condición a la reproducción, distribución y modificación de una obra. También se omite la atribución obligatoria del autor en todas las licencias CC. La licencia permite a los titulares de derechos marcar sus obras como de dominio público en tantos sistemas legales como sea posible.

Figure 11. CC0 Icon

Tomemos como ejemplo un documento de aprendizaje desarrollado con fines didácticos. El autor o autores pueden utilizar la etiqueta CC0 para garantizar que este material pueda ser utilizado por cualquier persona sin restricciones: puede copiarse total o parcialmente, distribuirse gratuitamente o mediante pago e integrarse total o parcialmente en otros documentos. Los autores originales no tienen que ser nombrados en ninguna parte.

Para identificar obras que no están sujetas a restricciones de derechos de autor, Creative Commons recomienda la *Marca de Dominio Público*:

> La Marca de dominio público funciona como una etiqueta, lo que permite a instituciones como éstas y otras con ese conocimiento comunicar que una obra ya no está restringida por derechos de autor y puede usarse libremente. por otros.
>
> — Creative Commons, Marca de dominio público

Figure 12. Public Domain Mark Icon

Selección de licencias y marcado de obras

Ya se ha comentado que Creative Commons pretende ser lo más sencillo posible tanto para los licenciantes (creadores de contenidos) como para los licenciatarios (destinatarios de las obras). En concreto, la organización ofrece mucha ayuda, desde la selección de la licencia adecuada hasta el etiquetado de la obra y su uso.

El *CC License Chooser* en el sitio web de la organización conduce paso a paso a la recomendación de la licencia adecuada planteando preguntas simples que un autor responde en relación con el

uso deseado de su trabajo (CC License Chooser).

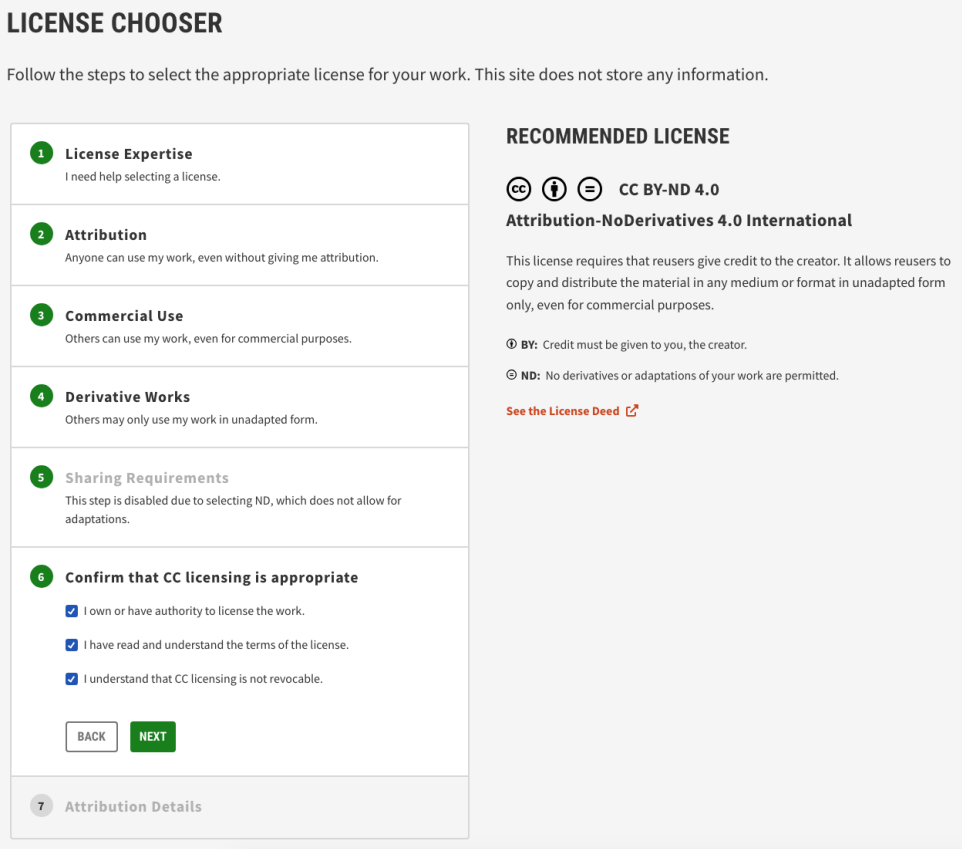

Figure 13. CC License Chooser

Si la licencia propuesta corresponde a las intenciones del autor, podrá marcar su obra en consecuencia. El Selector de licencia proporciona código HTML, por ejemplo, que el autor puede agregar directamente a un sitio web que muestra el trabajo (<<license-code> >).

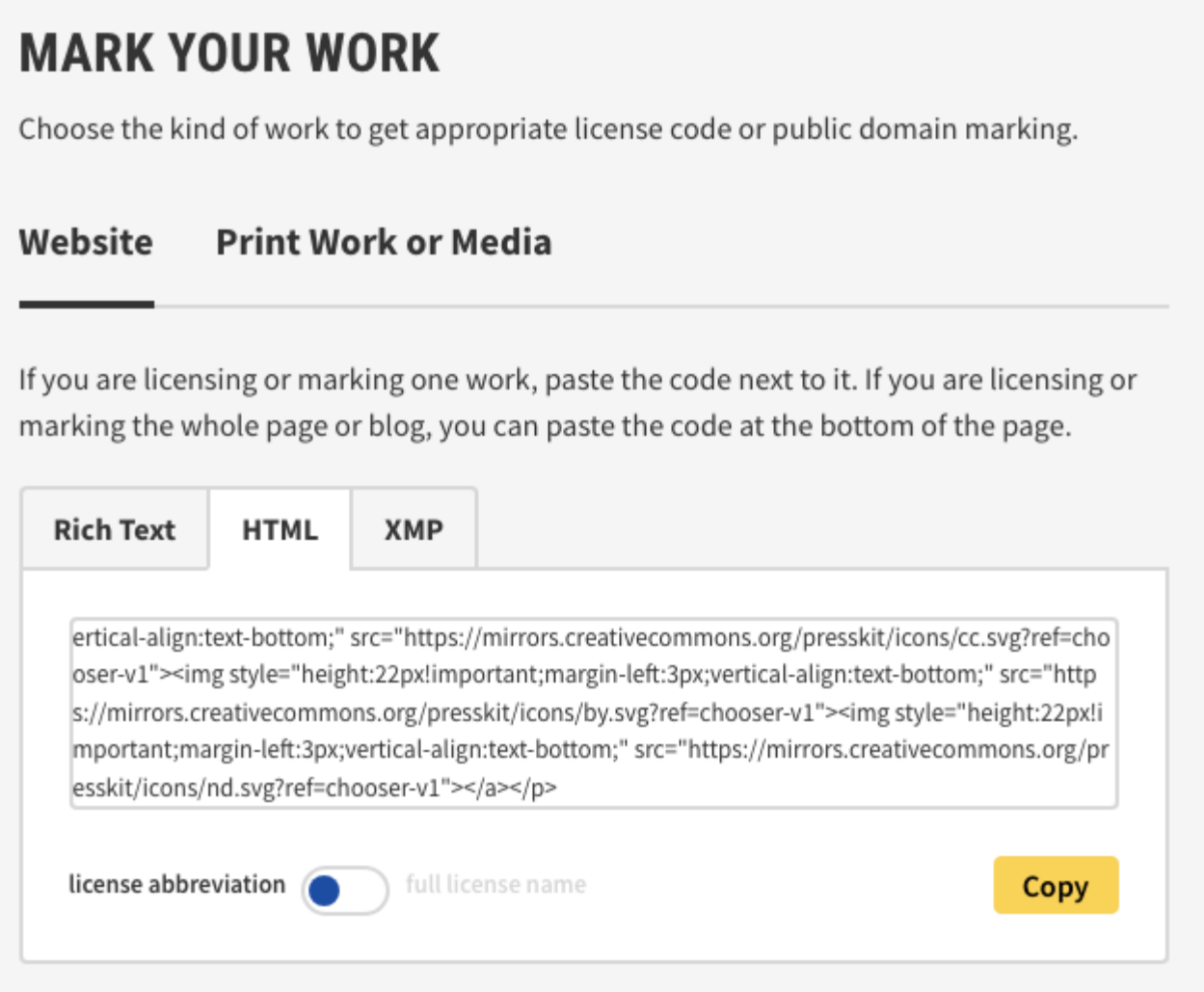

Figure 14. HTML code with link to the license

El resultado es entonces un aviso estandarizado con el enlace a la licencia correspondiente (License notice with link to the license).

Figure 15. License notice with link to the license

Ya se han introducido los iconos específicos de la licencia, en los que los módulos incluidos son directamente visibles. Se han consolidado como “llamativos” al identificar contenidos gratuitos en Internet.

Licencias internacionales y portadas

Desde el principio, Creative Commons se ha centrado en el desarrollo de licencias lo más generales posible, es decir, válidas en todo el mundo, lo que identifica con la adición de "internacional" (antes también "unported"). Por otro lado, existen variantes que tienen en cuenta las peculiaridades de un sistema jurídico regional o nacional y se denominan "ported". Por ejemplo, además de la licencia *CC BY-SA 3.0 Unported*, también existe una versión especial para Alemania con el nombre *CC BY-SA 3.0 Alemania*.

Con la versión 4.0 actualmente vigente de las licencias CC, Creative Commons se esfuerza por lograr una mayor estandarización y (hasta ahora) ha prescindido por completo de las versiones portadas. En la versión 4.0 se recomienda expresamente la licencia "internacional":

> Le recomendamos que utilice una licencia internacional versión 4.0. Esta es la versión más actualizada de nuestras licencias, redactada después de una amplia consulta con nuestra red global de afiliados, y ha sido escrita para que tenga validez internacional. Actualmente no hay puertos de 4.0 y está previsto que se creen pocos, si es que se crea alguno.
>
> — Creative Commons, Preguntas frecuentes

Ejercicios guiados

1. ¿Qué módulo del sistema de licencias Creative Commons forma parte de las seis licencias Creative Commons?

2. ¿Qué módulo del sistema de licencias Creative Commons exige la distribución de una obra en las mismas condiciones?

3. ¿En qué se diferencia CC0 de las seis licencias principales de CC?

4. ¿Cuál es la diferencia entre licencias “internacionales” y “portadas”?

Ejercicios exploratorios

1. ¿Es posible colocar una fotografía de una obra que es de dominio público bajo una licencia CC?

2. ¿Puede un autor publicar su novela, en la que incorpora un soneto de Shakespeare en su totalidad, bajo una licencia CC?

3. Un usuario publica una foto suya en su sitio web bajo una licencia CC. ¿Puede impedir la difusión de esta imagen invocando sus derechos personales?

Resumen

Fundada en 2001, la organización Creative Commons tiene como objetivo apoyar el intercambio y el uso de obras creativas, es decir, obras sujetas a derechos de autor, con la ayuda de herramientas legales gratuitas. La atención se centra en cuatro módulos (Atribución, No comercial, NoDerivs y ShareAlike), que se combinan en seis licencias principales y pueden ser seleccionadas por los autores para sus trabajos.

Respuestas a ejercicios guiados

1. ¿Qué módulo del sistema de licencias Creative Commons forma parte de las seis licencias Creative Commons?

   El módulo “Atribución”, abreviado “BY.”

2. ¿Qué módulo del sistema de licencias Creative Commons exige la distribución de una obra en las mismas condiciones?

   El módulo “Share Alike”, abreviado como “SA.”

3. ¿En qué se diferencia CC0 de las seis licencias principales de CC?

   Con CC0, un autor no garantiza ciertos derechos, como ocurre con las otras licencias CC, sino que renuncia a *todos* los derechos sobre la obra dentro del alcance de las posibilidades de la jurisdicción respectiva. Este principio de “no hay derechos reservados” corresponde a la designación de la obra como *dominio público*.

4. ¿Cuál es la diferencia entre licencias “internacionales” y “portadas”?

   Hasta la versión 3.0, Creative Commons ofrecía variantes de sus licencias que tenían en cuenta las particularidades de una jurisdicción regional o nacional. Desde 4.0, CC ha prescindido de estas versiones llamadas “portadas” y recomienda la versión “internacional” válida globalmente de la licencia respectiva.

== Respuestas a ejercicios exploratorios

5. ¿Es posible colocar una fotografía de una obra que es de dominio público bajo una licencia CC?

   Esto depende de si la fotografía cumple criterios según los cuales está protegida por derechos de autor. En otras palabras, la fotografía en sí debe ser reconocible como una obra creativa y digna de protección, por ejemplo a través de la perspectiva, el fondo, los filtros, etc. Si este es el caso, se puede colocar bajo una licencia CC y las condiciones de la CC. Luego solicite la licencia. Sin embargo, la parte original, es decir, el motivo real, sigue siendo de dominio público.

6. ¿Puede un autor publicar su novela, en la que incorpora un soneto de Shakespeare en su totalidad, bajo una licencia CC?

   Sí, pero sólo las partes creativas del autor de la novela están sujetas a la licencia CC elegida por el autor. El soneto, que es de dominio público, sigue siendo de dominio público.

7. Un usuario publica una foto suya en su sitio web bajo una licencia CC. ¿Puede impedir la difusión de esta imagen invocando sus derechos personales?

   Al publicar una fotografía bajo una licencia CC, el propietario generalmente acepta su distribución. Se alcanza un límite cuando el uso de la fotografía viola los derechos personales del propietario, por ejemplo cuando la imagen se distorsiona gravemente o se utiliza con fines publicitarios o en un contexto político sin el consentimiento del propietario. Los derechos personales del titular no se ven afectados por la regulación de los derechos de uso por la licencia CC, por lo que en tales casos puede emprender acciones legales contra el uso de su fotografía.

053.3 Otras licencias de contenido abierto

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 053.3

Peso

1

Áreas de conocimiento clave

- Comprender las licencias para la documentación
- Comprensión de las licencias para conjuntos de datos y bases de datos
- Comprender los derechos otorgados por las licencias de contenido abierto
- Comprender las obligaciones creadas por las licencias de contenido abierto

Lista parcial de archivos, términos y utilidades

- Licencia de documentación libre GNU, versión 1.3 (GFDL)
- Licencia de base de datos abierta Open Data Commons (ODbL)
- Acuerdo de licencia de datos comunitarios – Permisivo, versión 1.0 (CDLA)
- Acuerdo de licencia de datos comunitarios – Uso compartido, versión 1.0 (CDLA)
- Acceso abiertos

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	053 Licencias de contenido abierto
Objectivo:	053.3 Otras licencias de contenido abierto
Lección:	1 de 1

Introducción

En lecciones anteriores ya se ha introducido el concepto de contenido abierto, centrándose en las licencias Creative Commons. Sin embargo, además de esas licencias, existen otras de contenido abierto, algunas de las cuales se relacionan con contenido específico y también se utilizan regularmente para “subproductos” (como documentación) de proyectos de código abierto.

Licencias para documentación (software)

Los grandes repositorios de software contienen periódicamente documentación o manuales del software además del código. La documentación suele contener varios elementos protegidos por derechos de autor, como textos explicativos, gráficos ilustrativos o ejemplos de código. Las licencias de código abierto que se aplican al contenido de un repositorio, generalmente si emplean a dichos textos no hay otra información disponible; por ejemplo, GPLv3.0 se aplica a “cualquier trabajo sujeto a derechos de autor” y no se limita al código.

Sin embargo, la documentación suele estar sujeta a diferentes condiciones de licencia. Esto puede tener sentido por varias razones, porque la documentación se produce y utiliza de manera diferente al código. Las condiciones de licencia de una licencia copyleft estricta para programas

de ordenador pueden impedir la adopción de partes individuales de la documentación para otros programas. Tampoco está claro qué se entiende por proporcionar el "código fuente" para la documentación con licencia GPLv3.0. Además, solía ser una práctica común distribuir versiones impresas de documentación o manuales de programas. Esta circunstancia se puede tener mejor en cuenta mediante licencias de documentación específicas (por ejemplo, si algunas obligaciones de licencia están vinculadas a un cierto número de copias distribuidas).

Las licencias Creative Commons se suelen utilizar para la documentación, pero también existen licencias copyleft especiales para la documentación, como la *Free Documentation License* (FDL), que explica su finalidad en su preámbulo:

> El propósito de esta Licencia es hacer que un manual, libro de texto u otro documento funcional y útil sea "libre" en el sentido de libertad: asegurar a todos la libertad efectiva de copiar y redistribuir, con o sin modificarlo, ya sea comercial o no comercialmente. En segundo lugar, esta Licencia preserva para el autor y el editor una forma de obtener crédito por su trabajo, sin ser considerado responsable por las modificaciones realizadas por otros.
>
> Esta Licencia es una especie de "copyleft", lo que significa que los trabajos derivados del documento deben ser libres en el mismo sentido. Complementa la Licencia Pública General GNU, que es una licencia copyleft diseñada para software libre.
>
> Hemos diseñado esta Licencia para utilizarla en manuales de software libre, porque el software necesita documentación: un programa libre debe venir con manuales que proporcionen las mismas libertades que el software. Pero esta Licencia no se limita a los manuales de software; se puede utilizar para cualquier trabajo textual, independientemente del tema o de si se publica como libro impreso. Recomendamos esta Licencia principalmente para trabajos cuyo propósito sea instrucción o referencia.
>
> — Licencia de documentación libre, Preámbulo

La licencia es muy específica en cuanto a documentación; por ejemplo, también aborda las versiones impresas. En particular, una licencia de documentación copyleft no afecta explícitamente al código con el que se entrega. Aunque también sería posible distribuir documentación bajo GPLv3.0 junto con código con licencia MIT: Dado que en este caso serían dos trabajos independientes, la GPLv3.0 de la documentación no supondría que el código tuviera que tener licencia GPLv3.0. Pero una licencia dedicada a la documentación proporciona más claridad a este respecto.

Licencias para Bases de Datos

También existen licencias especiales para bases de datos. Estas licencias tienen en cuenta, entre otras cosas, el uso de una base de datos como recurso para un programa y las particularidades de

la protección de derechos de autor de las bases de datos.

A qué se aplica el término “Base de datos”

En la legislación sobre derechos de autor, en algunos ordenamientos jurídicos (incluido el europeo) se hace una distinción entre obras de bases de datos que, al igual que otras obras protegidas por derechos de autor (imágenes, textos, etc.), gozan de protección sobre la base de la *creación intelectual personal* (selección y disposición de elementos), y bases de datos para cuya producción (adquisición, verificación, presentación) se requirió una *inversión sustancial* (derecho de base de datos).

En ambos casos, el objeto de protección es una colección de elementos independientes (como obras o datos distintos) que están ordenados sistemática o metódicamente y son accesibles individualmente por vía electrónica o por otros medios. La protección de los derechos de autor de los elementos en sí no es importante, por lo que también pueden ser meros valores numéricos.

Por lo tanto, una colección de datos puede protegerse mediante derechos de autor como una obra de base de datos o bajo un derecho de base de datos (en la UE). Para que una obra de base de datos esté protegida, la disposición y selección de los elementos debe expresar una creación intelectual personal. Es poco probable que este sea el caso de las recopilaciones de datos que se utilizan como base para el software, pero pueden protegerse bajo el derecho de base de datos. Para ganar este derecho se debe haber realizado una inversión sustancial en la recolección y disposición de los elementos. Esta inversión también puede consistir en tiempo y recursos humanos.

Actualmente hay disponibles varias licencias para bases de datos y colecciones de datos que cubren mejor el uso de información que las licencias de software de código abierto “clásicas”. Echemos un vistazo a la distinción usando un ejemplo.

Como parte de un proyecto de investigación académica, todos los poemas de un autor deben recopilarse y publicarse en un sitio web, donde están ordenados alfabéticamente por título y se puede acceder a ellos individualmente. Sin embargo, el proyecto requiere que los derechos de autor pertinentes para la distribución y reproducción de los poemas se obtengan del autor. Para ello se pagan derechos de licencia.

Los derechos de licencia pueden constituir una “inversión sustancial”. Dado que los poemas se pueden recuperar individualmente y están ordenados sistemáticamente (según el alfabeto), la base de datos que contiene los poemas se puede proteger bajo el derecho de base de datos. Sin embargo, los elementos (poemas) no fueron seleccionados específicamente (la selección se hizo basándose en su integridad) y la disposición no es creativa, sino puramente sistemática. Por lo tanto, no existe ninguna creación intelectual personal en relación con la base de datos, por lo que

la base de datos no se considera una obra de base de datos.

Si, por el contrario, los poemas hubieran sido ordenados y seleccionados según ciertos criterios — por ejemplo, para documentar ciertos motivos de la obra del autor en el contexto del período creativo — podría existir una creación intelectual personal, y entonces esta colección de poemas se podría proteger como trabajo de base de datos.

Sin embargo, las dos categorías reciben un trato diferente según la ley de derechos de autor, en particular con respecto al plazo de protección: las bases de datos bajo el derecho de base de datos están protegidas sólo 15 años después de su producción/publicación, a diferencia de 70 años después de la muerte del autor para las obras de bases de datos.

Delimitación de datos

Los *datos* contenidos (en nuestro ejemplo, los poemas) siempre deben considerarse por separado de la base de datos en su conjunto, porque pueden (pero no necesariamente) ser elegibles para protección por derecho propio y, por lo tanto, pueden estar sujetos a diferentes condiciones de licencia. Por ejemplo, la colección de poemas puede estar sujeta a la Licencia de base de datos abierta (ver más abajo), mientras que los poemas individuales pueden estar sujetos a una licencia Creative Commons o una licencia de propiedad.

Delimitación para el Sistema de Gestión de Bases de Datos (DBMS)

Además, el software para gestionar los datos, el llamado *sistema de gestión de bases de datos* (DBMS), que no se considera parte de la base de datos según la ley de derechos de autor, sino un medio para acceder a los datos, debe considerarse por separado según la ley de derechos de autor. Estas aplicaciones, como MySQL, PostgreSQL, MariaDB y otras, deben considerarse por separado como programas informáticos y, como tales, se pueden proteger.

Conjuntos de datos para entrenar modelos de aprendizaje automático

Se pueden descargar varias colecciones de datos de sitios web como huggingface.co o kaggle.com, por ejemplo para entrenar modelos de aprendizaje automático. Estas colecciones, también conocidas como *conjuntos de datos*, suelen estar sujetas al derecho de base de datos, siempre que se haya realizado una "inversión sustancial" en su producción (lo que normalmente no es evidente desde el exterior).

El término "conjunto de datos" es algo vago porque puede referirse a una línea en una base de datos (es decir, una entrada o un elemento definible) pero también a colecciones de datos o conjuntos de datos.

Licencia de base de datos abierta (ODbL)

Estas distinciones también son visibles en la *Open Database License* (ODbL), una licencia ampliamente utilizada en el sector de las bases de datos. El ODbL fue desarrollado por Open Data Commons, un proyecto de Open Knowledge Foundation. La licencia distingue claramente en su preámbulo entre protección de la base de datos y protección del contenido individual de la base de datos. La licencia se refiere únicamente a lo primero, mientras que los contenidos también pueden licenciarse en otros lugares independientemente de esto.

Por supuesto, todavía es posible elegir la misma licencia para la base de datos y el contenido, por ejemplo, compilar una colección de obras bajo licencia CC-BY-4.0 en una base de datos que a su vez tenga licencia bajo CC-BY-4.0.

La mejor licencia para la base de datos depende de los objetivos del productor de la base de datos. Siguiendo con nuestro ejemplo de la colección de poesía: si la base de datos de poesía debe modificarse a voluntad y también puede distribuirse bajo otras condiciones de licencia, el ODbL no sería la elección correcta. Esto se debe a que el ODbL contiene un copyleft para bases de datos, lo que significa que una base de datos derivada de una base de datos con licencia ODbL (por ejemplo, una base de datos que adopta todos los elementos y la disposición de los elementos de la base de datos original y agrega otros) puede redistribuirse sólo bajo las mismas condiciones. En este caso, una licencia Creative Commons con derechos de edición (por ejemplo, CC-BY) es una mejor opción.

La ODbL también deja claro que de ninguna manera se aplica a los programas informáticos que gestionan la base de datos (es decir, un DBMS): “Esta licencia no se aplica a los programas informáticos utilizados en la creación u operación de la base de datos.”

Sin embargo, la licencia estipula que una obra creada a partir del contenido de la base de datos y disponible para uso público (la llamada *obra producida*) debe al menos indicar qué base de datos (no DBMS) se utilizó para ella y que la base de datos en sí puede ser utilizado según las condiciones de la ODbL. Un ejemplo de tal *trabajo producido* es un mapa de calles creado a partir de las coordenadas recopiladas en la base de datos.

Si para una base de datos se utilizara una de las licencias copyleft conocidas (específicas del software), como por ejemplo GPLv3.0, se puede suponer que un programa informático que utilice la base de datos (por ejemplo, para un sistema de tienda) sólo podría distribuirse bajo los términos de GPLv3.0. Por otra parte, es probable que la licencia de un DBMS sea independiente de la licencia de la base de datos porque el DBMS no se considera un “trabajo derivado” de esta; simplemente permite el acceso y la edición y, por tanto, es una obra independiente.

Acuerdos de licencia de datos comunitarios (CDLA)

Los *Acuerdos de licencia de datos comunitarios* (CDLA), un proyecto de la Fundación Linux, también se centra en la concesión de licencias de datos (a diferencia del software). Las recopilaciones de datos de capacitación para sistemas de aprendizaje automático también pueden ser objeto de CDLA.

> Nuestras comunidades querían desarrollar acuerdos de licencia de datos que pudieran permitir el intercambio de datos similar a lo que tenemos con el software de código abierto. El resultado es una colaboración a gran escala sobre licencias para compartir datos bajo un marco legal que llamamos Acuerdo de Licencia de Datos Comunitarios (CDLA).
>
> Estas licencias establecen el marco para el intercambio colaborativo de datos que hemos visto que funciona en comunidades de software de código abierto.
>
> — sitio web de CDLA (cdla.dev)

Por lo tanto, la CDLA se basa en un marco que tiene en cuenta diferentes niveles de cuestiones de licencia, especialmente cuando los datos se compilan de múltiples fuentes por diferentes contribuyentes. En nuestro ejemplo, podría ser una colección en la que diferentes autores aportan sus propios poemas para crear una base de datos de poesía común.

CDLA distingue entre la "licencia entrante" para los datos que se agregan a la colección y la "licencia saliente" para el uso de los datos. En un tercer nivel, especifica los requisitos para la organización que aloja o conserva los datos.

Siguiendo con el ejemplo de la poesía: un autor que aporta su poema se guiará por las especificaciones de "licencias entrantes". La "licencia de salida" se refiere a las condiciones acordadas por los editores de la colección de poesía en su conjunto.

Al igual que las licencias de software de código abierto, las CDLA están disponibles en las categorías *permisiva* y *compartida*, que presentaremos brevemente. Una visita a los sitios web kaggle.com o huggingface.co antes mencionados le dará una idea de qué licencias se utilizan para bases de datos o conjuntos de datos: por ejemplo, puede usar filtros para descubrir qué colecciones de datos se ofrecen bajo CDLA.

Acuerdo de licencia de datos comunitarios: permisivo

La versión 2.0 de la permisiva CDLA está disponible desde 2021 y es mucho más concisa y claramente formulada que la versión 1.0.

La versión 1.0 otorga derechos para usar y publicar los datos bajo licencia, incluida la publicación de los llamados "datos mejorados", que también incluyen ediciones o adiciones al conjunto de

datos. Al transmitir o publicar los datos se deben respetar las obligaciones básicas de licencia; entre otras cosas, se debe transmitir el texto de la licencia, se deben marcar los cambios y se deben conservar los avisos de derechos de autor. Los datos también se pueden transmitir bajo condiciones de licencia adicionales o de otro tipo.

En la versión 2.0, la expresión “datos mejorados” ya no se utiliza; en cambio, la concesión de derechos se formula de manera más precisa:

> Un Destinatario de datos puede usar, modificar y compartir los Datos puestos a disposición por los Proveedores de datos en virtud de este acuerdo si ese Destinatario de datos sigue los términos de este acuerdo.
>
> — Acuerdo de licencia de datos comunitarios - Permisivo, versión 2.0

La única condición para compartir los datos en la versión 2.0 es la transmisión del texto de la licencia o un enlace al texto de la licencia.

Acuerdo de licencia de datos comunitarios: compartir

La versión compartida de CDLA, que actualmente sólo está disponible en la versión 1.0, contiene un copyleft para los datos de tal manera que la transferencia (“publicación”) de “datos mejorados” sólo se permite bajo las mismas condiciones.

> Los datos (incluidos los datos mejorados) deben publicarse en virtud de este Acuerdo de conformidad con esta Sección 3;
>
> — Acuerdo de licencia de datos comunitarios - Compartir, versión 1.0

Acceso abierto

Un término que se utiliza a menudo en relación con el contenido abierto es *acceso abierto*. Aunque el término no está definido en el sentido legal, la declaración de la Iniciativa de Acceso Abierto de Budapest (BOAI) deja clara la intención detrás del acceso abierto. La iniciativa surgió de una conferencia en Budapest en 2001 y resume los esfuerzos internacionales por el acceso abierto:

> La literatura que debería ser de libre acceso en línea es aquella que los académicos dan al mundo sin expectativa de pago. Principalmente, esta categoría abarca sus artículos de revistas revisados por pares, pero también incluye cualquier preimpresión no revisada que deseen publicar en línea para comentar o alertar a sus colegas sobre hallazgos de investigación importantes. Hay muchos grados y tipos de acceso más amplio y más fácil a esta literatura. Por “acceso abierto” a esta literatura nos referimos a su libre disponibilidad

> en la Internet pública, lo que permite a cualquier usuario leer, descargar, copiar, distribuir, imprimir, buscar o vincular los textos completos de estos artículos, rastrearlos para indexarlos, pasarlos como datos a un software o utilizarlos para cualquier otro fin lícito, sin barreras financieras, legales o técnicas distintas de las inseparables del acceso a Internet. La única limitación a la reproducción y distribución, y el único papel del derecho de autor en este ámbito, debería ser dar a los autores control sobre la integridad de su trabajo y el derecho a ser reconocidos y citados adecuadamente.
>
> — Declaración de la Iniciativa de Acceso Abierto de Budapest

Por tanto, el acceso abierto se refiere en particular al acceso gratuito a la literatura científica y otros contenidos en Internet. El movimiento de acceso abierto se originó en la década de 1990 con el objetivo de poner las publicaciones científicas a disposición del público en general. Por lo tanto, el acceso abierto no se refiere a una licencia específica o a determinadas condiciones de concesión de licencia, sino más bien a un concepto de concesión de licencia.

Cuando un artículo se publica en acceso abierto, el primer paso es elegir una licencia de contenido abierto, como por ejemplo una de las licencias Creative Commons. Sin embargo, la elección de una licencia CC no es obligatoria para el acceso abierto. A lo largo de los años, se ha desarrollado una categorización de diferentes estrategias de acceso abierto, que también tienen en cuenta la doble licencia: por ejemplo, otorgar una licencia completa a un editor y al mismo tiempo el autor ofrece el contenido para descargar en su propio sitio web.

Tanto la BOAI como la *Declaración de Berlín* que siguió dos años después son hitos importantes en el movimiento de acceso abierto. El preámbulo de la Declaración de Berlín establece:

> De acuerdo con el espíritu de la Declaración de la Iniciativa de Acceso Abierto de Budapest, la Carta de ECHO y la Declaración de Bethesda sobre Publicaciones de Acceso Abierto, hemos redactado la Declaración de Berlín para promover Internet como un instrumento funcional para una base mundial de conocimientos científicos y una reflexión humana y especificar medidas que los responsables de las políticas de investigación, las instituciones de investigación, las agencias de financiación, las bibliotecas, los archivos y los museos deben considerar.
>
> — Declaración de Berlín 2003

Ejercicios guiados

1. ¿Se pueden utilizar también las licencias de código abierto “clásicas” para documentación y bases de datos?

Sí	
No	
Depende, no hay una respuesta clara	

2. ¿Tiene sentido utilizar licencias independientes para la documentación? ¿Por qué?

3. ¿Tiene sentido utilizar licencias independientes para bases de datos? ¿Por qué?

4. ¿Cuáles de las siguientes licencias son adecuadas para la documentación?

CC-BY-4.0	
FDL	
ODbL	
GPL-3.0	
BSD-3-Clause	

5. ¿Qué se entiende por el término “acceso abierto”?

Ejercicios exploratorios

¿Existen licencias específicas para fuentes?

+

1. Explique brevemente dos estrategias o modelos de acceso abierto.

2. ¿Qué se entiende por “definición abierta”?

Resumen

Además de las conocidas licencias Creative Commons, existen otros conceptos de licencia de contenidos que se adaptan aún más específicamente a los respectivos tipos de datos: para documentación de software o bases de datos, están disponibles las licencias FDL u ODbL que tienen en cuenta las características de este tipo de obras. Esta lección proporciona una descripción general de algunas de estas licencias y también aborda el concepto de acceso abierto.

Respuestas a ejercicios guiados

1. ¿Se pueden utilizar también las licencias de código abierto “clásicas” para documentación y bases de datos?

Sí	X
No	
Depende, no hay una respuesta clara	

2. ¿Tiene sentido utilizar licencias independientes para la documentación? ¿Por qué?

   Sí, porque suelen contener otros contenidos además del propio programa. También regulan cómo se deben procesar, por ejemplo, las impresiones de la documentación.

3. ¿Tiene sentido utilizar licencias independientes para bases de datos? ¿Por qué?

   Sí, puede resultar útil. En particular, una licencia de base de datos específica puede lograr un efecto copyleft para la base de datos sin que el copyleft afecte el código circundante. Además, las licencias de bases de datos abordan disposiciones específicas que la ley de derechos de autor establece para la protección de las bases de datos.

4. ¿Cuáles de las siguientes licencias son adecuadas para la documentación?

CC-BY-4.0	X
FDL	
ODbL	
GPL-3.0	
BSD-3-Clause	

5. ¿Qué se entiende por el término “acceso abierto”?

   El término describe el acceso libre a literatura científica y otros contenidos en Internet.

Respuestas a ejercicios exploratorios

1. ¿Existen licencias específicas para fuentes?

   Sí, existe, por ejemplo, la licencia SIL Open Font License (OFL), que contiene un copyleft específico para las fuentes, pero esto no tiene ningún efecto si las fuentes se utilizan sin cambios.

2. Explique brevemente dos estrategias o modelos de acceso abierto.

   En la estrategia de acceso abierto “Gold”, el editor pone a disposición la publicación directamente bajo una licencia de contenido abierto, generalmente (pero no exclusivamente) utilizando licencias Creative Commons.

   El modelo de acceso abierto “verde” se aplica cuando el editor no distribuye la publicación en acceso abierto, pero el autor tiene la oportunidad de hacer que la publicación esté disponible para su descarga, por ejemplo, en su sitio web bajo una licencia abierta.

3. ¿Qué se entiende por “definición abierta”?

   La “definición abierta” es una comprensión común, formulada por la Open Knowledge Foundation, del término “abierto” en “datos abiertos”, “conocimiento abierto” y “contenido abierto”. " La definición describe en particular las libertades básicas que deben concederse para que una licencia sea “abierta” en el sentido de la definición. Estos incluyen acceso, uso, edición o modificación y uso compartido.

Linux Professional Institute

Tema 054: Modelos de negocio de código abierto

054.1 Modelos de negocio de desarrollo de software

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 054.1

Peso

2

Áreas de conocimiento clave

- Comprender los objetivos y razones para publicar software o contenido bajo una licencia abierta
- Comprender los modelos comerciales comunes y las fuentes de ingresos para las organizaciones que desarrollan software de código abierto y contenido abierto
- Comprender las implicaciones del uso de software de código abierto como componentes de productos y servicios tecnológicos más amplios
- Comprender el impacto que tienen las licencias en los modelos de negocio de desarrollo de software
- Comprender las consideraciones del software de código abierto desde la perspectiva del cliente
- Conciencia de las estructuras de costos y las inversiones necesarias para los modelos comerciales de desarrollo de software de código abierto

Lista parcial de archivos, términos y utilidades

- Desarrollo pagado
- Complementos de núcleo abierto y pagos
- Freemium
- Versiones empresariales y comunitarias

- Distribución autohospedada
- Suscripciones
- Atención al cliente

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	054 Modelos de negocio de código abierto
Objectivo:	054.1 Modelos de Negocio de Desarrollo de Software
Lección:	1 de 1

Introducción

Tradicionalmente, las empresas que distribuían software propietario (también conocido como software de código cerrado) tenían un modelo de negocios relativamente sencillo: vendían una licencia para usar el software, ya sea como una tarifa única o de forma continua en un modelo de suscripción.

Pero a lo largo de las décadas, el mercado de software ha cambiado radicalmente y hoy los clientes esperan pagar unos pocos dólares por una aplicación y recibir actualizaciones gratuitas de por vida.

Como resultado, el enfoque de la vieja escuela hacia el software propietario ya no es rentable, e incluso las empresas que solían construir todo su modelo de negocio en torno a la venta de licencias de software se han diversificado hacia la integración de proveedores, soporte, servicios y software como servicio. (SaaS), alojamiento en la nube/contenedores y productos de hardware que ejecutan software (teléfonos, computadoras portátiles, servidores, impresoras, automóviles, relojes inteligentes, etc.).

Por el contrario, las licencias de software libre y de código abierto otorgan a cualquiera el derecho a utilizar, estudiar, modificar y redistribuir el software sin pagar una tarifa. Por lo tanto, las empresas basadas en código abierto nunca tuvieron la opción de vender una licencia para utilizar el software. No debería esperar simplemente tomar el trabajo de otros de un proyecto de código abierto y obtener grandes ganancias al entregar ese trabajo disponible públicamente a sus clientes, porque ellos pueden obtener el mismo software de forma gratuita directamente desde el proyecto. En cambio, las empresas que ofrecen software libre y de código abierto utilizan modelos de negocio alternativos. Los modelos de negocio de software más exitosos suelen combinar múltiples formas de valor para el cliente.

Esta lección analiza por qué una empresa elegiría algunos de estos modelos de negocio de software con software de código abierto y las ventajas y desventajas de estas opciones.

Objetivos y razones para lanzar software o contenido bajo una licencia abierta

Hay muchas razones por las que los desarrolladores eligen construir un negocio con código fuente abierto: el software puede satisfacer una necesidad, resolver un problema o proporcionar algunas funciones que los desarrolladores ofrecen a los clientes. Una empresa puede ahorrar tiempo y dinero compartiendo el trabajo de software con otros desarrolladores, en lugar de escribir el mismo software desde cero. Participar en el proyecto es una inversión que, con suerte, dará sus frutos en un software utilizable, estable, confiable, seguro y bien mantenido.

Muchos desarrolladores de código abierto esperan obtener correcciones de errores de sus clientes. Algunos clientes llevan sus contribuciones a un nivel aún mayor al agregar nuevas funciones, migrar el software a un nuevo entorno o realizar otras mejoras significativas. Tanto las correcciones de errores como las mejoras de nivel superior mejoran el software original. Si los clientes los devuelven a los desarrolladores originales, los cambios podrían hacer que el código sea más deseable para otros clientes potenciales y fomentar su adopción.

Pero una empresa no puede esperar contribuciones y atención simplemente colocando su código en un sitio como GitHub o GitLab. Una cosa es animar a los desarrolladores a unirse a un proyecto, pero es más difícil mantenerlos entusiasmados con él, así que no subestimes la cantidad de trabajo que requieren las contribuciones de la comunidad. Una empresa debe esforzarse en reclutar desarrolladores, orientarlos, motivarlos y verificar si su código tiene fallas.

Otra razón para habilitar el código es crear un estándar industrial. Compartir código de manera abierta podría conducir a una mejor interoperabilidad y otros beneficios. El equipo que desarrolló el código tiene una experiencia valiosa que podría permitirles dirigir el futuro del proyecto y recibir un pago por apoyar a otros que utilizan el código.

Algunas empresas abren su código para generar confianza entre los clientes. En primer lugar, los clientes saben que pueden seguir usando y mejorando el código si la empresa fracasa o decide ingresar a un mercado diferente y abandona el código. En segundo lugar, los clientes pueden comprobar la calidad y seguridad del código por sí mismos o pedirle a un experto independiente que realice una auditoría, lo que normalmente no es posible con software cerrado y propietario.

Finalmente, una empresa podría haber adoptado un código base que ya es de código abierto y planear construir un negocio comercial sobre este. La licencia puede requerir que la empresa distribuya el código fuente de sus cambios a cualquier usuario final del programa.

En definitiva, algunas razones para abrir el código de una empresa son:

- Aprovechar un proyecto de software libre existente
- Beneficiarse de la innovación y las contribuciones de los clientes
- Crear confianza entre los clientes
- Promover el código como estándar de la industria

Modelos de negocio y flujos de ingresos comunes

La industria del software, durante las últimas décadas, ha descubierto muchos modelos de negocio apropiados tanto para empresas propietarias como para empresas de código abierto. Esta sección se centra en los más populares que se utilizan actualmente entre los desarrolladores de código abierto.

Un modelo de negocio, común tanto en el software propietario como en el de código abierto, es cobrar por el soporte. Los clientes pueden tener problemas para instalar y configurar el software, corregir errores a medida que se encuentran, agregar nuevas funciones para su uso exclusivo y administrar sus sistemas. El personal que desarrolló el software es una excelente fuente de apoyo. De hecho, este personal probablemente ya esté ofreciendo soporte gratuito en foros donde se analiza el software.

Por lo tanto, en el modelo de soporte, un cliente paga a un proveedor para que instale el software de código abierto en el hardware del cliente (llamado distribución de autohospedaje) u obtiene acceso al software en el hardware del proveedor (un modelo basado en la nube). Los contratos de soporte garantizan que los clientes obtengan ayuda oportuna de la empresa para sus necesidades más complejas. Las empresas que utilizan este modelo generalmente ofrecen anticipos, que los clientes pagan de forma regular.

Otro modelo de negocio para software propietario — en realidad un conjunto de modelos superpuestos — se llama *freemium*, un término popularizado en 2009 por el autor y editor Chris

Anderson, y basado en un modelo de negocio mucho más antiguo conocido como *razor and blades*. En el modelo de afeitadora y hojas, los clientes pagan muy poco por el producto base (por ejemplo, una afeitadora o una impresora) y luego pagan una prima por los componentes necesarios diseñados para desgastarse (hojas para la afeitadora, cartuchos de tinta para la impresora). .

En el modelo freemium, los clientes pueden utilizar un producto en un determinado nivel sin coste alguno y se les anima a pagar por más funciones si el producto les resulta útil. Probablemente conozca sitios de noticias en línea que ofrecen una cierta cantidad de artículos gratuitos y solicitan a los lectores que se suscriban para obtener acceso completo a sus sitios; Este es un ejemplo bien conocido de modelo freemium. Otro ejemplo es una plataforma de juegos que ofrece el juego base de forma gratuita, pero cobra dinero por un inventario específico.

Otras empresas de software propietario basan su modelo freemium en el tiempo: prueba el software gratis durante tres meses y luego paga para seguir usándolo.

Las empresas de software propietario a veces utilizan una versión del modelo freemium conocida como *open core*. En el núcleo abierto, ciertas funciones básicas (el “núcleo” del producto) son de código abierto y se pueden licenciar funciones patentadas adicionales.

A menudo, la parte abierta es completamente funcional, pero funciona mejor para investigadores o usuarios individuales, y se vuelve difícil de administrar cuando muchas personas la comparten en una empresa. Por lo tanto, las características propietarias pueden incluir interfaces web convenientes para la administración, herramientas de contabilidad y colaboración, y otras cosas de particular interés para sitios grandes.

Aunque los modelos de negocio centrales abiertos utilizan algún software de código abierto, los clientes son lo suficientemente inteligentes como para reconocer que todo el producto es en realidad propietario. Entonces, si bien el núcleo abierto puede tener la ventaja de basarse en un proyecto de código abierto existente, no proporciona ninguna otra ventaja del software de código abierto. Un modelo de negocio central abierto no genera confianza en los clientes, no los inspira a contribuir al software base, no les brinda acceso para examinar el código, no construye una red de desarrolladores capacitados fuera de la empresa principal ni funciona como un estándar de la industria. Peor aún, el modelo de desarrollo de núcleo abierto tiene los mismos costos de desarrollo, sin los beneficios que lo hacen valer la pena. Las empresas que prueban un modelo de negocio central abierto generalmente quedan decepcionadas con los resultados y muchas luego cambian a un modelo propietario puro.

Las empresas también pueden crear un servicio web sobre una base de código abierto o propietario, en el modelo de software como servicio (SaaS). Los clientes pagan mensualmente o anualmente.

Muchas empresas que ejecutan servicios web SaaS u ofrecen aplicaciones móviles ganan dinero mediante publicidad. Algunas empresas también recopilan datos sobre los usuarios a través del servicio o la aplicación y los venden a terceros que pueden utilizarlos con fines publicitarios. Sin embargo, los anuncios pueden parecer molestos. La venta de datos es controvertida y algunos países imponen restricciones a la recopilación de datos.

Finalmente, las empresas podrían desarrollar y lanzar software de código abierto sin tratar de obtener ningún ingreso de ello. Este modelo está disponible para empresas que obtienen ingresos de otros servicios además del software. Por ejemplo, las empresas automotrices colaboran para crear grandes cantidades de software libre para ejecutar en sus automóviles (que hoy en día están completamente informatizados).

Las principales empresas de software que obtienen sus ingresos de ofertas patentadas, como Google y Amazon, a veces lanzan software administrativo u otras herramientas útiles como código abierto, porque estas herramientas de código abierto no son fundamentales para su negocio principal. Las organizaciones que publican el código bajo una licencia abierta se benefician de los comentarios, informes de errores y mejoras de funciones proporcionadas por la comunidad de código abierto.

Uso de software de código abierto en otras tecnologías y servicios

Muchas empresas incorporan software de código abierto en sus productos o plataformas web. Después de todo, ¡el software de código abierto es gratuito! Pero esa no es la razón más importante (y quizás ni siquiera una buena razón) para adoptarlo. Más importante aún, gran parte de este software es de alta calidad (aunque el equipo de desarrollo debería examinarlo cuidadosamente antes de adoptarlo) e incluso puede ser un estándar de la industria.

Otra ventaja del software de código abierto es que seguirá estando disponible si los desarrolladores o la comunidad que lo rodea desaparecen.

Pero el software gratuito y de código abierto conlleva responsabilidades. Esta sección enumerará rápidamente las principales cuestiones a considerar antes de adoptarlo.

La primera cuestión se aplica a cualquier software o herramienta de terceros que se esté considerando para su adopción: ¿satisfará las necesidades de los usuarios ahora y a medida que la empresa evolucione? ¿El software cuenta con el respaldo de una comunidad sólida que pueda ofrecer soporte técnico y continuar desarrollando el software? ¿El software sufre importantes vulnerabilidades de seguridad?

A continuación, los directivos deben considerar las responsabilidades de la empresa si incorpora

un proyecto de código abierto en su propio software. Si los desarrolladores crean código nuevo en torno al código fuente abierto, deben verificar qué deben hacer según la licencia del software de código abierto. Algunas licencias requieren que los desarrolladores distribuyan el código fuente de sus propios cambios a sus usuarios finales bajo la misma licencia gratuita que el código base original.

Incluso si un desarrollador no está obligado a distribuir su código fuente modificado y está creando un producto propietario sobre el código fuente abierto, es posible que el desarrollador quiera contribuir con ciertas cosas, como correcciones de errores y mejoras al código fuente abierto. Al contribuir con estas correcciones y mejoras, el desarrollador colaborador permite que el proyecto las incorpore (si los desarrolladores principales así lo deciden) y las mantenga. Es posible que los desarrolladores que no aporten mejoras tengan que volver a aplicar las correcciones y mejoras cada vez que actualicen a una nueva versión del código original.

Debido a esta dependencia, y por otras razones, el personal de la empresa debería considerar convertirse en miembros activos de la comunidad que desarrolla el código. Los desarrolladores pueden aprender mucho sobre el código al participar y pueden ayudar a establecer la dirección para el desarrollo futuro. Por supuesto, la organización debería pagar por el tiempo que los desarrolladores dedican a actividades comunitarias. Para una empresa que se toma en serio el uso de software libre, no es gratuito.

Consideraciones del software de código abierto desde la perspectiva del cliente

Abrir código es muy beneficioso para los clientes por varios motivos, pero implica una relación diferente entre la empresa y sus clientes. Los clientes deben comprender los beneficios y las implicaciones de la relación.

El principal beneficio para el cliente, mencionado anteriormente en esta lección, es que puede confiar más en el software. Saben que no desaparecerá. Muchas empresas propietarias cierran o despegan repentinamente en una nueva dirección, abandonando a los clientes con quizás poco tiempo para migrar a un producto que tal vez no les guste. El software de código abierto, por el contrario, no depende de una sola organización. Si el proyecto es importante, otras personas de la comunidad continuarán con él cuando los desarrolladores originales se muden.

Los clientes también pueden comprobar la calidad y seguridad del software de código abierto. Pueden determinar con qué facilidad pueden agregar funciones propias o trasladarlas a un nuevo entorno.

Debido a que el código fuente de un proyecto de código abierto está disponible para que todos lo examinen, una amplia gama de desarrolladores pueden familiarizarse con él. En un proyecto que

tiene una comunidad activa, muchas personas brindan apoyo en el foro. A menudo, es fácil contratar personas para dar soporte al software o para administrarlo y utilizarlo dentro de la empresa, porque los nuevos empleados ya conocen el código.

Es muy tranquilizador para los clientes, que dependen del código para su funcionamiento diario, saber que pueden corregir un error ellos mismos o contratar a alguien para que lo haga. Un error oscuro que afecta sólo a unos pocos clientes puede tardar años en ser solucionado por el equipo central de desarrollo, una frustración constante para los usuarios de software propietario. Además, cuando el código fuente está disponible, es posible identificar más rápidamente un error y su solución sugerida.

¿Qué pasa con la relación entre la empresa y el cliente? La empresa puede optar por establecer una relación exactamente similar a la que ofrece una típica empresa de software propietario, proporcionando al cliente actualizaciones periódicas y un contrato de soporte. El cliente nunca necesita mirar el código fuente ni unirse a los foros de la comunidad.

Pero la mayoría de los clientes se beneficiarían de las oportunidades únicas que ofrecen los proyectos de código abierto. Pueden permitir que sus propios desarrolladores contribuyan tanto con información como con código. Dicha participación profundiza la comprensión de su personal, les ayuda a reclutar nuevo personal y les da voz en el desarrollo futuro.

Estructuras de Costos e Inversiones

Los programadores tienen una gran demanda, por lo que cualquier esfuerzo de software será costoso. Las personas que conocen los principales proyectos de código abierto tienen aún más demanda, porque esas bases de código se utilizan ampliamente.

Un proyecto de código abierto ofrece posibles ahorros de costos porque diferentes organizaciones e individuos pueden combinar sus esfuerzos en una base de código común. Pero ejecutar un proyecto de este tipo introduce nuevos costos.

Si una empresa abre un proyecto desarrollado internamente, debe invertir para que sirva a una comunidad más grande (posiblemente mundial). Es posible que sea necesario repensar algunas funciones que satisfacían las estrechas necesidades comerciales de la empresa para que sirvieran también a otras empresas.

Si el código es deficiente o incómodo, la empresa probablemente no debería abrirlo. Tanto los contribuyentes potenciales como los clientes potenciales se sentirán repelidos por la calidad. Corresponde a los desarrolladores solucionar estos problemas de todos modos, porque el software mal codificado es frágil: es difícil actualizarlo con nuevas funciones y tiende a desarrollar errores complejos que son difíciles de solucionar. Estos problemas se denominan *deuda técnica* y cuanto

antes se solucionen, mejor será para todos los usuarios.

Finalmente, es sorprendente la frecuencia con la que una empresa ha incorporado secretos comerciales, contraseñas, referencias personales a individuos u otra información confidencial en el código. Los desarrolladores tienen que invertir tiempo en eliminar esas vulnerabilidades. De todos modos, las contraseñas, las claves API, los certificados y las credenciales de acceso a la nube nunca deberían estar en el código; deben gestionarse externamente a través de un servicio seguro.

Digamos que una empresa ha abierto su código y espera beneficiarse de contribuciones externas. Algunos clientes pagarán a los desarrolladores por el mantenimiento y las nuevas funciones. Otros pondrán a sus propios desarrolladores en el proyecto, pero el equipo central de desarrollo tiene que dedicar tiempo para apoyarlos. El equipo central de desarrollo necesita educar a los externos sobre el código y los estándares de codificación asociados. Este equipo también debe guiar a los contribuyentes externos sobre qué agregar y dónde enviar comentarios sobre su código.

Esté atento a los forasteros que muestran talento. Estas personas podrían ser reclutas valiosos para el equipo central. Es posible que les guste unirse al equipo y cuentan con un conocimiento considerable.

Cuando a un equipo de desarrolladores se le paga, mientras otras personas ofrecen su tiempo como voluntarias, los voluntarios pueden sentirse explotados o preguntar por qué deberían ayudar. Cada contribuyente, ya sea un voluntario individual o una organización, debe someterse a los tipos de pensamiento descritos anteriormente en esta lección para decidir por qué está contribuyendo.

El código libre no es gratuito, aunque no implique costes de licencia. Es simplemente un modelo diferente de desarrollo.

Ejercicios guiados

1. ¿Cómo mejora el código fuente abierto la confianza del cliente en el software?

2. ¿Por qué las empresas se sienten tentadas a probar un modelo de negocio central abierto y por qué el modelo no logra ofrecer los beneficios del software de código abierto?

3. ¿Qué es la distribución autohospedada?

4. ¿Cuáles son los tipos típicos de ayuda que los desarrolladores brindan a los contribuyentes externos a sus proyectos de código abierto?

Ejercicios exploratorios

1. Está considerando basar un producto propietario en un proyecto de código abierto. ¿Qué factores consideraría para tomar la decisión?

2. Ha creado varios productos, por los que cobra una suscripción, además de un proyecto de código abierto. ¿Qué hará si la licencia de código abierto requiere que contribuya con todo su código al proyecto? Además, agregó soporte para algunos protocolos de comunicación nuevos al proyecto de código abierto para satisfacer sus necesidades de propiedad. Si tiene la opción, ¿le pedirá al proyecto de código abierto que integre este soporte de protocolo en su código central?

Resumen

En esta lección, aprendió el valor de abrir el código fuente. Usted revisó los modelos comerciales comunes que se utilizan hoy en día y la importancia de comprender el impacto de la licencia del código. Lea sobre cuestiones a considerar al incorporar código fuente abierto en su empresa y cómo el código abierto beneficia a sus clientes. También aprendió en qué se diferencian los costos del desarrollo de código abierto de los del software propietario.

Respuestas a ejercicios guiados

1. ¿Cómo mejora el código fuente abierto la confianza del cliente en el software?

   Los clientes pueden comprobar la calidad y seguridad del código. También pueden confiar en que podrán seguir utilizando el código si los desarrolladores originales dejan de admitirlo.

2. ¿Por qué las empresas se sienten tentadas a probar un modelo de negocio central abierto y por qué el modelo no logra ofrecer los beneficios del software de código abierto?

   A primera vista, parece que el núcleo abierto debería ofrecer todos los beneficios del software de código abierto, al tiempo que permite a una empresa seguir utilizando un modelo de negocio propietario que vende una licencia para utilizar el software. En la práctica, un modelo de negocio central abierto no logra ofrecer las ventajas del código abierto y, al mismo tiempo, sigue teniendo todos los mayores costos del desarrollo abierto.

3. ¿Qué es la distribución autohospedada?

   Distribución autohospedada significa una versión de software que se puede ejecutar en el propio equipo del cliente.

4. ¿Cuáles son los tipos típicos de ayuda que los desarrolladores brindan a los contribuyentes externos a sus proyectos de código abierto?

   Los desarrolladores de un proyecto normalmente educan y asesoran a los contribuyentes externos y verifican que sus contribuciones cumplan con los estándares de codificación y calidad del equipo.

Respuestas a ejercicios exploratorios

1. Está considerando basar un producto propietario en un proyecto de código abierto. ¿Qué factores considerarías para tomar la decisión?

   Primero, decida si el software de código abierto satisface sus necesidades. Verifique su calidad, las fallas de seguridad que se hayan reportado públicamente y la salud de la comunidad que lo rodea. Verifique la licencia cuidadosamente para ver si requiere que comparta sus modificaciones al código. Determina cuánto deseas participar en la comunidad del proyecto de código abierto y cómo definirás los roles que tus desarrolladores desempeñarán en esa comunidad.

2. Ha creado varios productos, por los que cobra una suscripción, además de un proyecto de código abierto. ¿Qué hará si la licencia de código abierto requiere que contribuya con todo su código al proyecto? Además, agregó soporte para algunos protocolos de comunicación nuevos al proyecto de código abierto para satisfacer sus necesidades de propiedad. Si tiene la opción, ¿le pedirá al proyecto de código abierto que integre este soporte de protocolo en su código central?

   Si el proyecto no requiere que compartas cambios de código, probablemente no lo harás porque puedes cobrar una suscripción más fácilmente por un producto propietario. Si tiene que compartir su código, ofrezca una suscripción a una distribución autohospedada. Incluso si no necesita compartir su código, probablemente quiera pedirle al proyecto que integre su soporte para protocolos de comunicación para que no tenga que volver a implementar ese soporte cada vez que instale una nueva versión del código abierto. código.

054.2 Modelos de negocio de proveedores de servicios

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 054.2

Peso

2

Áreas de conocimiento clave

- Comprender los modelos comerciales comunes y las fuentes de ingresos para las organizaciones que brindan servicios relacionados con software de código abierto y contenido abierto
- Comprender el impacto que tienen las licencias en los modelos de negocio de un proveedor de servicios
- Comprender los objetivos de nivel de servicio y los acuerdos de nivel de servicio
- Comprender la necesidad de protección de la seguridad y la privacidad
- Conciencia de las estructuras de costos y las inversiones necesarias para los modelos comerciales de servicios de software de código abierto

Lista parcial de archivos, términos y utilidades

- Servicios alojados
- Nubes
- Consultoría
- Capacitaciones
- Ventas de hardware
- Soporte al usuario

- Términos de servicio (ToS)
- Objetivos de nivel de servicio (SLO)
- Acuerdos de nivel de servicio (SLA)
- Acuerdos de procesamiento de datos

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	054 Modelos de negocio de código abierto
Objectivo:	054.2 Modelos de negocio de proveedores de servicios
Lección:	1 de 1

Introducción

Los proveedores de servicios son la columna vertebral de cualquier empresa u organización que opere de cualquier forma a través de una red. Desde el proveedor de servicios de Internet (ISP), que proporciona acceso a Internet global, hasta aplicaciones especializadas como el correo electrónico o la gestión de clientes, son los servicios basados en software los que hacen posibles las operaciones comerciales reales.

El software de código abierto es un componente de muchos modelos comerciales diferentes de proveedores de servicios. Por ejemplo, un sistema operativo de código abierto podría constituir la base de una empresa de servicios de alojamiento. Una herramienta de orquestación de código abierto podría ser la columna vertebral de una empresa de infraestructura como servicio (IaaS) o plataforma como servicio (PaaS), como un proveedor de nube pública o una plataforma de contenedores. Una aplicación de código abierto podría entregarse en línea como software como servicio (SaaS). Algunos proveedores de servicios también ofrecen servicios complementarios relacionados con el software de código abierto, como consultoría, formación o atención al cliente.

Flujos de ingresos

Cuando el servicio se basa entera o en gran parte en software de código abierto, podemos llamarlo *modelo de negocio de código abierto*. Dado que el software de código abierto se puede descargar y utilizar sin pagar una tarifa, los modelos comerciales de servicios de código abierto han desarrollado productos y servicios específicos dependiendo, entre otras cosas, de la proporción de software libre en el producto real.

Si todo el software está bajo una licencia libre, un modelo de negocio puede consistir en *hosting*, es decir, proporcionar el software en una plataforma confiable y segura. Esto elimina el esfuerzo y el riesgo de que los clientes operen sus propios servidores. A cambio, los clientes pagan tarifas de uso al proveedor de servicios. Las tarifas pueden estar relacionadas con la cantidad de cuentas de usuario, el volumen de datos que se almacenan o transfieren, o ciertos intervalos de tiempo. En algunos casos, varios proveedores diferentes ofrecen servicios pagos para el mismo software de código abierto.

Este popular modelo de negocio a menudo implica *suscripciones*, donde los clientes pagan una tarifa mensual o anual para acceder al servicio. El acceso suele combinarse con servicios o funciones adicionales. Los proveedores pueden ofrecer múltiples niveles de suscripciones, cobrando una tarifa más alta por más funciones, más usuarios, servicios más completos o una mayor garantía de confiabilidad. Algunos proveedores también pueden ofrecer un nivel básico del servicio de forma gratuita, una variación del modelo “freemium”.

Otra fuente de ingresos popular para el modelo de negocio de proveedores de servicios de código abierto siempre ha sido ofrecer *soporte*. El soporte incluye asesoramiento, documentación y capacitación para los usuarios que tienen problemas al utilizar el software. Si el cliente prefiere alojar el software libre en sus propios servidores, la instalación en particular a menudo puede resultar difícil porque es posible que el usuario necesite instalar otras herramientas y bibliotecas de soporte, puede que no sepa dónde colocarlas para que todas las piezas funcionen juntas, o pueden tener problemas en su hardware y sistema operativo particulares. Por lo tanto, el apoyo y la formación de un proveedor de servicios es valioso.

También se pueden desarrollar y proporcionar *servicios adicionales*, como características funcionales o relacionadas con la seguridad de forma específica para el cliente. Estas características o adaptaciones del software a los requisitos de un cliente son una adición sofisticada y lucrativa al modelo de negocio. Corresponde al proveedor de servicios y al cliente decidir si las agregaciones vuelven al proyecto básico como software libre o se convierten en software propietario.

Algunos proyectos de código abierto ofrecen un *modelo de licencia dual*. El software está disponible bajo una licencia de software libre, que satisface las necesidades de muchos usuarios.

Pero algunos usuarios quieren incorporar el software a un producto propietario, lo que no se puede hacer bajo una licencia recíproca. Por lo tanto, se proporciona a estos usuarios una licencia estándar basada en derechos de autor. El modelo de licencia dual es más eficaz para las bases de datos, porque a muchas empresas les gustaría ofrecer una base de datos eficiente y rica en funciones como parte de un producto de software propietario.

En un *modelo de ingresos por publicidad*, los clientes no pagan por el acceso al servicio. En cambio, el proveedor de servicios muestra anuncios a los clientes que utilizan el servicio y cobra a las empresas por colocar sus anuncios. Los proveedores de servicios de código abierto a menudo evitan el uso de servicios publicitarios propietarios debido a preocupaciones sobre la privacidad del usuario. Sin embargo, los servicios publicitarios alternativos de código abierto no realizan un seguimiento de anuncios invasivo.

En un *modelo basado en comisiones*, el proveedor de servicios recibe un porcentaje o tarifa por gestionar las transacciones a través de su servicio en línea. Este modelo es la base de la mayoría de los sitios de comercio electrónico, sitios de reserva de viajes y procesadores de pagos. Muchos proyectos de código abierto en este espacio son plataformas de propósito general destinadas a ser autohospedadas por el proveedor de servicios u ofrecidas como un servicio secundario basado en comisiones, donde el sitio web de cara al cliente paga un porcentaje o tarifa por una versión alojada del plataforma de código abierto.

Un *modelo de negocio de contenido abierto* implica la creación y distribución de contenido bajo licencias como Creative Commons para que otros puedan usarlo, modificarlo y compartirlo libremente. Se anima a los usuarios a contribuir al contenido, mejorándolo con el tiempo. Si bien el contenido en sí es libre, se pueden generar ingresos a través de productos o servicios complementarios, como donaciones, mostrar anuncios, ofrecer funciones freemium, vender productos relacionados o brindar servicios de consultoría, capacitación o soporte.

Elegir entre servicios alojados (hosted) y autohospedados (self-hosted)

Los servicios alojados son ideales para organizaciones que buscan facilidad de uso, menores costos iniciales, escalabilidad y gestión de seguridad profesional sin la necesidad de conocimientos técnicos profundos. Sin embargo, estos servicios presentan posibles desventajas en términos de control, personalización, privacidad de datos y dependencia del proveedor.

Los servicios alojados suelen requerir una configuración mínima y el proveedor de servicios se encarga del mantenimiento, las actualizaciones y la seguridad del sistema. A menudo ponen a disposición centros de datos globales, lo que proporciona un mejor rendimiento y confiabilidad para los usuarios internacionales. Los proveedores de servicios pueden aumentar o reducir los recursos según la demanda con un mínimo esfuerzo por parte del cliente. Además, los servicios alojados suelen tener equipos de seguridad dedicados a gestionar y responder a las amenazas, y

los proveedores de servicios pueden ofrecer cumplimiento con diversos estándares regulatorios.

Por otro lado, los servicios alojados almacenan los datos confidenciales de los clientes fuera de las instalaciones, lo que puede generar preocupaciones sobre la privacidad y la confianza de los datos. Un entorno de alojamiento compartido puede plantear riesgos de seguridad adicionales. Las interrupciones o el tiempo de inactividad del proveedor de servicios afectan directamente la disponibilidad del servicio del cliente. Depender de un solo proveedor puede resultar problemático si el proveedor cambia los términos o interrumpe el servicio.

Por lo tanto, los servicios autohospedados son adecuados para organizaciones con la experiencia técnica y los recursos necesarios para gestionar su infraestructura. Sin embargo, el autohospedaje implica costos iniciales más altos, mantenimiento continuo y desafíos de escalabilidad. La elección entre servicios alojados y autohospedados depende en última instancia de las necesidades, las capacidades técnicas, el presupuesto y las prioridades de control, personalización y privacidad de los datos de cada cliente.

Impacto de las licencias

Básicamente, las licencias de software de código abierto funcionan bien con los modelos de negocio de los proveedores de servicios porque el cliente no espera ser propietario del software que proporciona el servicio. En cambio, pagan por el acceso al servicio.

Por otro lado, el uso de software de código abierto en servicios como IaaS, SaaS o PaaS plantea cuestiones legales que muchas veces no se aclaran. Sólo unas pocas licencias, en particular la *Licencia Pública General GNU Affero*, regulan explícitamente el uso de software libre “en el caso del software de servidor de red”.

Con otras licencias populares (como las restrictivas copyleft, la Licencia Pública General GNU y las permisivas tipo BSD), hay margen de discreción en cuanto a lo que se puede considerar “copiar”, “transmitir” o “distribución” cuando el software se utiliza a través de la red. Las decisiones sobre si el código fuente se pondrá a disposición de la aplicación o los requisitos de atribución, y en qué forma, dependen de dichas definiciones. También debe aclararse legalmente la conexión con componentes de software propietarios o la concesión de licencias de componentes complementarios de desarrollo propio. Por estas razones, es crucial en el contexto de los modelos de negocio de los proveedores de servicios aclarar las cuestiones de licencia relacionadas con el uso de software libre.

Es una buena práctica que los proveedores de servicios enumeren públicamente todo el software de código abierto que utilizan junto con su licencia respectiva, incluso cuando no estén obligados legalmente a hacerlo.

Consideraciones de seguridad y protección de la privacidad

El éxito de un modelo de negocio de proveedor de servicios depende en gran medida de una experiencia satisfactoria para el cliente, especialmente con software de código abierto, donde los clientes tienen la libertad de alojar el software ellos mismos o elegir un proveedor de servicios para el mismo software. Por lo tanto, la principal ventaja competitiva de un proveedor de servicios de código abierto es la experiencia del cliente que ofrece, que puede ser más conveniente, rentable, confiable, segura o eficaz que el autohospedaje. Sin embargo, la contrapartida es que los clientes deben compartir datos con el proveedor de servicios, lo que puede generar problemas de privacidad para datos personales o sensibles.

Por seguridad, los clientes deben asegurarse de que el proveedor de servicios tenga un proceso sólido para aplicar rápidamente parches de seguridad. Vale la pena evaluar cómo el servicio gestiona las dependencias de otros proyectos de código abierto para garantizar que todo el software esté actualizado y sea seguro. El cliente también debe evaluar las políticas de seguridad del proveedor, incluida la forma en que manejan el cifrado de datos, los controles de acceso y la respuesta a incidentes.

La transparencia del software de código abierto permite una revisión exhaustiva del código por parte de la comunidad, lo que puede mejorar la seguridad al identificar y corregir vulnerabilidades. Por lo tanto, los clientes deben buscar revisiones o auditorías del software realizadas por terceros acreditados o expertos en seguridad dentro de la comunidad. Estas auditorías deben confirmar que el proveedor sigue prácticas y estándares de codificación segura durante el proceso de desarrollo, y deben considerar si el proveedor utiliza canales de CI/CD con controles de seguridad integrados para detectar vulnerabilidades de manera temprana.

Para la privacidad, los clientes deben verificar que el servicio recopile y almacene solo los datos necesarios para su funcionamiento, minimizando el riesgo de exposición de los datos. El servicio debe cifrar los datos tanto en tránsito como en reposo, mediante un cifrado seguro. Los mecanismos de control de acceso deben garantizar que sólo el personal autorizado pueda acceder a datos confidenciales.

El cliente debe comprobar que el servicio ofrece a los usuarios control sobre sus datos, como la posibilidad de eliminarlos o exportarlos. El servicio debe cumplir con las regulaciones de privacidad relevantes, como el Reglamento General de Protección de Datos (GDPR) de la Unión Europea y la Ley de Privacidad del Consumidor de California (CCPA), y brindar transparencia sobre sus prácticas de manejo de datos.

Es importante verificar que el proveedor tenga políticas y procedimientos claros para responder a las violaciones de datos, incluidos los requisitos de notificación. El cliente también debe revisar la política de privacidad del proveedor para comprender cómo se recopilan, utilizan, comparten y

protegen los datos, y considerar a los terceros con los que el proveedor pueda compartir datos y asegurarse de que también cumplan con estrictos estándares de privacidad.

En general, es valioso observar los comentarios y reseñas de la comunidad para evaluar la reputación y confiabilidad del software y del proveedor de servicios. Una comunidad u organización sólida debe mantener y apoyar activamente el proyecto de código abierto. El cliente debe verificar que el proveedor tenga procedimientos regulares de respaldo de datos y planes sólidos de recuperación ante desastres, y verificar si el servicio utiliza redundancia de datos para garantizar la disponibilidad y confiabilidad.

Los proveedores de servicios deben ser conscientes de que los clientes evaluarán sus prácticas de seguridad y privacidad, la reputación de la comunidad, el cumplimiento de las regulaciones y la transparencia en el manejo de datos y, en consecuencia, deben tomar medidas para seguir las mejores prácticas.

Acuerdos comunes entre el proveedor de servicios y el cliente

Los proveedores de servicios suelen adoptar términos y acuerdos legales que forman la columna vertebral legal y operativa de la relación entre la empresa y sus clientes. Estos acuerdos ayudan a garantizar una comunicación clara, establecer expectativas, definir responsabilidades y brindar protección legal para ambas partes.

Los *Términos de servicio* (ToS) de un servicio, también conocidos como *Términos y condiciones* (T&C) o *Términos de uso*, son un acuerdo legal entre un proveedor de servicios y los clientes o usuarios de ese servicio. Los ToS describen las reglas, responsabilidades y limitaciones que rigen el uso del servicio. Protege tanto al proveedor de servicios como al cliente al definir claramente lo que cada parte puede y no puede hacer mientras proporciona o utiliza el servicio. Algunos elementos comunes de un acuerdo ToS son la confirmación que el usuario acepta cumplir con sus términos, pautas sobre comportamiento aceptable y actividades prohibidas, detalles sobre quién es el propietario del contenido creado o subido por los usuarios, información sobre la creación, seguridad y terminación de la cuenta, arbitraje o procesos de mediación para la resolución de conflictos, y límites a la responsabilidad del prestador del servicio por daños y perjuicios.

Una *Política de Privacidad* es una declaración que describe cómo el proveedor de servicios recopila, utiliza, almacena y protege los datos del usuario. Algunos elementos comunes de una política de privacidad son los tipos de datos que el proveedor de servicios recopila y sus métodos de recopilación, cómo utiliza los datos, las condiciones bajo las cuales el proveedor de servicios puede compartir datos con terceros e información sobre los derechos de los usuarios con respecto a sus datos, tales como como acceso, rectificación y supresión.

Un *Acuerdo de nivel de servicio* (SLA) es un acuerdo formal y documentado entre un proveedor de

servicios y un cliente que describe el nivel de servicio esperado, incluidas las métricas de desempeño, responsabilidades y expectativas específicas. El SLA define los estándares para la prestación de servicios, proporcionando un marco claro para medir y gestionar el desempeño del servicio. Algunos elementos comunes de un SLA incluyen una descripción detallada de los servicios prestados, objetivos específicos para el desempeño del servicio, como tiempo de actividad y tiempos de respuesta, métodos para medir e informar el desempeño, consecuencias por no cumplir con los objetivos de desempeño y procedimientos para revisar y actualizar el SLA.
.

Un *Objetivo de Nivel de Servicio* (SLO) es un componente clave de un Acuerdo de Nivel de Servicio que especifica metas u objetivos medibles para el desempeño del servicio. Estos objetivos cuantifican el nivel esperado de servicio entre un proveedor de servicios y un cliente y se utilizan para garantizar que el servicio cumpla consistentemente con los estándares acordados. Algunos elementos comunes de los SLO incluyen métricas específicas que el proveedor de servicios medirá para evaluar el desempeño del servicio (por ejemplo, tiempo de actividad, tiempo de respuesta, tiempo de resolución y rendimiento), los valores deseados o esperados para cada métrica de desempeño, las técnicas y herramientas utilizadas para medir y monitorear las métricas de desempeño, los componentes o aspectos específicos del servicio que cubre el SLO (por ejemplo, hardware, software, componentes de red o procesos específicos) y las consecuencias o recompensas basadas en si el servicio cumple o no con los SLO (por ejemplo, sanciones financieras, créditos de servicio o bonificaciones por desempeño).

Un *Acuerdo de procesamiento de datos* (DPA) es un contrato entre un cliente (el controlador de datos) y un proveedor de servicios (el procesador de datos) involucrado en el procesamiento de datos personales. La DPA detalla las responsabilidades y obligaciones de ambas partes para garantizar el cumplimiento de las leyes de protección de datos, como el RGPD. En algunas jurisdicciones, es obligatorio un DPA entre un proveedor de servicios y un cliente.

Estructuras de costos e inversiones

Los gastos más importantes en un modelo de negocio de proveedor de servicios son los costos fijos relacionados con el alojamiento de los servicios, como la compra de hardware de servidor, el pago del espacio del centro de datos, la electricidad para alimentar y enfriar los servidores, el ancho de banda de la red y los salarios del personal de implementación y mantenimiento. Algunos proveedores de servicios también pueden tener costos variables por licencias de software o servicios de terceros. Los clientes que eligen autohospedar sus servicios asumen estos costos directamente.

Al elegir software de código abierto, los proveedores de servicios y los clientes pueden reducir o eliminar los costos de licencia en comparación con el software propietario, por definición, el software de código abierto no cobra ninguna tarifa por la licencia del software. El uso de software

de código abierto para crear servicios también puede ahorrar tiempo y recursos de desarrollo, y el mantenimiento y las actualizaciones impulsados por la comunidad pueden reducir los costos a largo plazo. Sin embargo, es importante considerar el costo total de propiedad (TCO) del software de código abierto, incluidos los posibles costos de soporte, mantenimiento e integración.

Ejercicios guiados

1. Mencione algunas formas en que los proveedores de servicios pueden obtener ingresos sin cobrar a los usuarios del servicio.

2. ¿Cuáles son algunas de las razones por las que un proveedor de servicios podría ofrecer su software para que los clientes lo autohospeden?

3. ¿Por qué un proveedor de servicios publicaría su software bajo la licencia GNU Affero GPL?

Ejercicios exploratorios

1. ¿Qué es una Política de Uso Aceptable (AUP)?

2. ¿Qué es un Acuerdo de licencia de usuario final (EULA)?

Resumen

Esta lección analiza los modelos comerciales de proveedores de servicios que se basan en software de código abierto y varias fuentes de ingresos, incluidas las suscripciones, los ingresos por publicidad, las tarifas por hora y los modelos basados en comisiones. Destaca el impacto de las licencias de software de código abierto en los proveedores de servicios, enfatizando sus beneficios y obligaciones. También se abordan las consideraciones de seguridad, privacidad y confiabilidad de los servicios, junto con la importancia de los Términos de servicio (ToS), los Acuerdos de nivel de servicio (SLA) y el Acuerdo de procesamiento de datos (DPA).

Respuestas a ejercicios guiados

1. Mencione algunas formas en que los proveedores de servicios pueden obtener ingresos sin cobrar a los usuarios del servicio.

   El servicio puede ofrecer anuncios por los que los anunciantes pagan. El servicio puede cobrar una comisión a los proveedores que ofrecen productos o servicios en el sitio, como comercio minorista o viajes. El proveedor de servicios puede vender datos de clientes, una práctica a la que muchos clientes se opondrían y que debería explicarse en los Términos de servicio.

2. ¿Cuáles son algunas de las razones por las que un proveedor de servicios podría ofrecer su software para que los clientes lo autohospeden?

   Los clientes potenciales pueden probar el software para comprobar sus características y confiabilidad antes de registrarse con el proveedor de servicios. Los clientes también pueden encontrar errores e incluso proponer soluciones.

3. ¿Por qué un proveedor de servicios publicaría su software bajo la licencia GNU Affero GPL?

   Affero GPL requiere que otros proveedores de servicios publiquen cualquier cambio que realicen en el software bajo la misma licencia. Esta disposición impide que los competidores se aprovechen del proveedor de servicios agregando algunas mejoras que mantienen como propietarias y luego promocionando su versión del servicio como superior a la original.

Respuestas a ejercicios exploratorios

1. ¿Qué es una Política de Uso Aceptable (AUP)?

   Una Política de Uso Aceptable (AUP) es una política que define usos aceptables e inaceptables del servicio, para evitar abusos y garantizar un entorno seguro y confiable. Algunos elementos comunes de una PUA incluyen una descripción de acciones específicas que no están permitidas, como spam o piratería, obligaciones de los usuarios de utilizar el servicio de manera responsable, acciones que el proveedor del servicio puede tomar en caso de violaciones, como suspender la cuenta de un usuario, procedimientos para informar y abordar violaciones.

2. ¿Qué es un Acuerdo de licencia de usuario final (EULA)?

   Un Acuerdo de licencia de usuario final (EULA) es un contrato legal entre un proveedor de software propietario y un usuario final (no una empresa u organización), que otorga al usuario el derecho a utilizar el software bajo ciertas condiciones. Algunos elementos comunes de los EULA incluyen el alcance y las limitaciones de la licencia (por ejemplo, uso personal o licencia no transferible), acciones prohibidas (por ejemplo, ingeniería inversa o redistribución), condiciones bajo las cuales se puede rescindir la licencia y la propiedad y protección de los derechos de propiedad intelectual. Los servicios de código abierto sustituyen una licencia de software de código abierto en lugar de un CLUF propietario.

054.3 Cumplimiento y mitigación de riesgos

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 054.3

Peso

3

Áreas de conocimiento clave

- Comprender cómo garantizar el cumplimiento de la licencia
- Comprender cómo mantener la información sobre las licencias
- Comprender el concepto de Oficinas de Programas de Código Abierto
- Comprender las implicaciones de los derechos de autor, las patentes y las marcas comerciales en los modelos comerciales de código abierto
- Conciencia de los riesgos legales relacionados con los modelos de negocio de código abierto
- Conciencia de los riesgos financieros relacionados con los modelos de negocio de código abierto

Lista parcial de archivos, términos y utilidades

- Análisis de composición de software (SCA)
- Lista de materiales del software (SBOM)
- Intercambio de datos de paquetes de software (SPDX)
- OWASP ciclónDX
- Oficinas de programas de código abierto (OSPO)
- La garantía del producto
- Responsabilidad del producto

- Regulaciones de exportación
- Impacto de fusiones y adquisicionesns

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	054 Modelos de negocio de código abierto
Objectivo:	054.3 Cumplimiento y mitigación de riesgos
Lección:	1 de 1

Introducción

Un viejo chiste sobre código abierto dice que “el software libre no viene gratis”. Si bien el software libre y de código abierto permite una gran innovación y creatividad, los usuarios y desarrolladores deben cumplir con una serie de reglas. Esta lección cubre el cumplimiento y el riesgo en el software de código abierto.

La lección enumera los pasos básicos que los desarrolladores deben cumplir con las licencias, los riesgos de utilizar software de código abierto y las formas de rastrear y catalogar el software que utiliza la organización. Veremos cómo estas tareas pueden convertirse en políticas y ser promovidas por una Oficina de Programas de Código Abierto (OSPO).

Requisitos para el lanzamiento de software basado en componentes de código abierto

Si la organización utiliza software internamente, las licencias de software libre y de código abierto no imponen requisitos. La organización puede definir sus propias reglas para evitar vulnerabilidades, asegurarse de que todos utilicen los mismos componentes o por otras razones.

Pero eso está más allá del alcance de esta lección. Cualquier requisito que las licencias de código abierto o la propia organización impongan al uso del software debe documentarse y la organización debe brindar capacitación sobre sus políticas.

Incluso si la organización ejecuta software en su servidor web y ofrece servicios con los que interactúan personas ajenas a la organización, la mayoría de las licencias de software libre y de código abierto no imponen requisitos. Sin embargo, la licencia pública GNU Affero también requiere el cumplimiento de los requisitos de copyleft para el software que se ejecuta como servicio.

Los requisitos legales de cada licencia importante de software libre y de código abierto se han explorado en profundidad en otras lecciones, por lo que esta sección solo ofrece una lista de acciones que los desarrolladores y administradores deben tomar para cumplir:

Examinar los componentes de código abierto

La organización necesita políticas claras y educación sobre qué software de código abierto utilizar y dónde incluirlo en los productos. Dichas políticas cubren el cumplimiento, así como otras cuestiones como garantizar la seguridad y participar en la comunidad que desarrolla el software.

Mantenga la licencia original en el código

Los desarrolladores no deben quitar la licencia del componente que utilizan. Incluir la licencia en el código fuente suele ser un requisito en la licencia. De hecho, quitar la licencia podría considerarse plagio, porque el desarrollador hace que parezca que él mismo escribió el código. Quitar la licencia también podría generar serios problemas más adelante, porque la organización podría violar la licencia cuando el código esté incluido en sus productos.

Llevar un registro de licencias

La organización debe mantener un catálogo que muestre la licencia de cada archivo o paquete utilizado por la organización, para asegurarse de que está cumpliendo con las licencias.

Reconocer al autor

Casi todas las licencias requieren que se le dé crédito al propietario de los derechos de autor (a menudo llamado “autor”) cuando se habla y promociona el software. Cada licencia explica qué incluir en este aviso: Normalmente, requieren un aviso de derechos de autor estándar con el año y el nombre del propietario de los derechos de autor. Este aviso debe aparecer en cualquier documentación relacionada con el producto que utiliza el componente, incluida la publicidad y el sitio web del dispositivo o programa.

No implique respaldo

Algunas licencias BSD requieren que la persona que distribuye un producto con los

componentes se asegure de no dar a entender que el propietario de los derechos de autor respalda el producto. Incluso si el requisito no está establecido, cumplirlo es lo ético.

Ofrecer el código fuente para licencias copyleft

Si una organización distribuye un archivo binario que contiene componentes copyleft, ya sea como distribución de software o en un dispositivo, la organización tiene que ofrecer al público el código fuente de esos componentes. Si la organización cambia el código y redistribuye el trabajo derivado en formato binario, el código fuente de la versión modificada (*derivada*) también debe ofrecerse al público. La distribución se puede realizar a través de cualquier medio que facilite el acceso al público, como publicar el código en un sitio web u ofrecerlo en un disco o memoria USB.

No incluya componentes copyleft en productos propietarios

Si la organización incorpora componentes copyleft en un producto, bajo algunas circunstancias la organización tiene que lanzar todo el producto bajo la misma licencia copyleft. Las condiciones que desencadenan este requisito varían de una licencia a otra. Sin embargo, a veces no está claro qué tipo de uso activa el requisito de copyleft. Entonces, excepto en situaciones claras como el uso de una biblioteca de código abierto (que no debería desencadenar el requisito recíproco del copyleft), los desarrolladores deben tener mucho cuidado con el uso de componentes copyleft a menos que la organización esté preparada para lanzar su producto bajo la misma licencia.

Cambios en el código del documento

Al distribuir versiones modificadas del código, indique claramente los cambios que realizó la organización.

Conceder derechos de patente

Algunas licencias de software libre o de código abierto requieren una concesión para el uso de la patente en el software. Por lo tanto, si una organización publica un código bajo dicha licencia y posee una patente sobre algún proceso en el código, la organización no puede cobrar una tarifa ni ejercer otros controles basados en su patente a nadie que use o adapte el código.

Respetar las marcas comerciales

Algunos software de código abierto están cubiertos por marcas comerciales. Las marcas comerciales pueden abarcar palabras y frases, logotipos y otras imágenes, y muchas otras cosas. Por un lado, una organización debe manejar adecuadamente la marca registrada para cualquier software que utilice: Una violación común es parodiar, alterar o simplemente mostrar una marca registrada sin permiso. Por otro lado, si la organización quiere utilizar la marca, debe cumplir con los requisitos del propietario de la marca, lo que podría descartar modificaciones en el software.

Riesgos del software de código abierto

Esta lección trata sobre el cumplimiento, por lo que nos centraremos en los riesgos en esa área, pero también cubriremos algunos otros temas.

Violación de licencia

Las licencias de software libre y de código abierto deben tratarse con la misma seriedad que otras licencias y contratos. Las organizaciones sí las hacen cumplir, como Software Freedom Conservancy, que actúa en nombre de pequeños proyectos con copyleft que no tienen sus propios mecanismos de aplicación.

Estas organizaciones suelen abordar las demandas como último recurso. La mayoría de las violaciones no son intencionadas y son fáciles de resolver con educación. Aún así, perjudica la reputación de una organización si se le considera ignorante e insensible hacia las comunidades de cuyo software depende.

Y de hecho, se han iniciado demandas cuando un usuario de software se niega flagrantemente a cooperar cuando el software y el demandado son lo suficientemente importantes.

Incluso si una organización no es castigada con una demanda, el daño que una violación de licencia causa a su relación con la comunidad, así como a su reputación, puede ser sustancial. Un proyecto puede descarrilarse por algo tan simple como que las preguntas de los desarrolladores queden sin respuesta en los foros dedicados al software que están intentando utilizar.

Puede ser un desastre para alguien encontrar software con copyleft en un producto propietario. Para cumplir con la licencia, los desarrolladores deben eliminar todo el código copyleft o liberar su producto bajo la licencia copyleft. Hacer que su software sea gratuito, con el código fuente disponible, podría tener efectos nefastos en los modelos de negocio basados en derechos de licencia u otras formas de monopolizar el valor del producto.

Confianza y reputación

Hemos visto que las violaciones de licencias pueden ser muy dañinas. Otros tipos de comportamiento que perturban la confianza y la reputación también introducen riesgos.

Algunas organizaciones lanzan software bajo una licencia de software libre o de código abierto y luego anuncian en algún momento que las versiones futuras estarán bajo una licencia propietaria. Este cambio puede resultar tentador, porque muchos proyectos de código abierto no reciben apoyo financiero de suficientes clientes.

Pero tales cambios de licencia provocan enojo tanto entre los clientes como entre los

desarrolladores externos que contribuyen al proyecto. A veces, los desarrolladores externos toman la versión libre y continúan desarrollándola de forma independiente, un paso conocido como *bifurcación* del proyecto. La versión libre podría volverse competitiva con la versión propietaria de la empresa y alejar a los clientes de la empresa.

También existen riesgos al participar en foros de proyectos. Una empresa debe aprender a ser un buen ciudadano. Los problemas comunes incluyen:

- Intentar utilizar las contribuciones financieras o de código de una empresa como palanca para forzar el proyecto en una dirección que otros desarrolladores no quieren.
- Hacer demasiadas exigencias a la comunidad, como presionarla con muchas solicitudes de funciones o incluso demasiadas preguntas.
- Ser grosero de otras maneras y violar el código de conducta del proyecto.
- Tratar de dominar la publicidad o capitalizar inapropiadamente de otras maneras el éxito del proyecto.

Inversiones no recompensadas

Los modelos de negocio se analizan en otra lección; Esta sección solo señala el riesgo financiero de participar en proyectos de código abierto.

Las empresas pueden iniciar o unirse a proyectos de código abierto con la expectativa de obtener ingresos a través de algún mecanismo, como contratos de soporte, contratos SaaS, recopilación de datos o incluso donaciones y subvenciones. Desafortunadamente, quienes llevan a cabo proyectos de código abierto a menudo los encuentran menos lucrativos de lo esperado. Por supuesto, cualquier empresa corre un riesgo al iniciar un nuevo proyecto, pero es particularmente difícil encontrar una fuente confiable de ingresos para el código abierto.

El código abierto parece más sostenible cuando respalda alguna otra forma de ingresos, como la venta de dispositivos de hardware, automóviles, impresoras, etc.

Bifurcaciones (Forks)

Como hemos visto, los miembros de un proyecto a veces no están de acuerdo con la hoja de ruta (roadmap), el liderazgo u otros aspectos del proyecto, por tanto, crean un proyecto alternativo en la misma base de código. Una organización también puede realizar la bifurcación para crear una versión especializada del software. Por ejemplo, Android utiliza una versión especializada de Linux que mantiene Google. (Google también hace muchas contribuciones a la versión principal de Linux y no siempre son aceptadas).

Las organizaciones pueden crear una bifurcación por varias razones. Es posible que envíen sus cambios al proyecto principal y los rechacen porque otros desarrolladores no los quieren. En un proyecto con una licencia permisiva, o un proyecto utilizado sólo dentro de la organización, es posible que desee mantener sus cambios en secreto.

El riesgo de una bifurcación es que los desarrolladores del proyecto bifurcado son responsables del mantenimiento de todo el producto. Si se agregan correcciones de errores importantes o nuevas características al proyecto principal, los desarrolladores del proyecto bifurcado deben duplicar los cambios o renunciar a sus beneficios. A medida que pasa el tiempo y los proyectos se alejan, ya que mantenerse al día con los cambios en el proyecto principal se vuelve cada vez más difícil.

Licencias incompatibles

Como se explicó en otras lecciones, algunas licencias de software libre y de código abierto tienen cláusulas incompatibles. Estos componentes no se pueden utilizar juntos en un producto.

Es probable que este problema surja durante una fusión o adquisición. Es posible que cada empresa haya estado utilizando software con una licencia particular y, si quieren combinar el código en sus proyectos, deben asegurarse de que las licencias sean compatibles.

Responsabilidad del producto

Muchas licencias de software de código abierto (y los términos de servicio de muchos software y servicios propietarios, de hecho) rechazan explícitamente cualquier responsabilidad por el uso del software. Otra forma de decir esto es que la licencia o los términos de servicio no ofrecen ninguna garantía.

Los proveedores de software rara vez son responsables de los problemas con su software, pero en ocasiones los clientes los demandan o intentan otras formas de castigo. Los tribunales no tienen que reconocer las cláusulas de “no garantía”.

Cada vez más, las leyes y regulaciones imponen responsabilidades a los creadores de software. Estas restricciones legales — o al menos, propuestas de restricciones — ahora se ven particularmente para la inteligencia artificial, pero a veces son más amplias.

Regulaciones de Exportación

Muchos países restringen la exportación de bienes y software. En el ámbito del software, dichas regulaciones se aplican con mayor frecuencia a los productos de seguridad y, específicamente, al uso de la criptografía. Estados Unidos solía tener controles estrictos sobre cualquier software que

contenga criptografía, pero los controles se han relajado en los proyectos de código abierto.

Las empresas deben conocer las regulaciones de exportación donde están creando software, en caso de que estas regulaciones afecten la venta y distribución de sus productos.

Lista de materiales del software: Conozca lo que está utilizando

Muchos productos vienen con una lista de materiales, que es solo una lista de todos sus componentes. Por ejemplo, cuando compra un producto de consumo, puede incluir una hoja de papel con una lista de todas las piezas, hasta las tuercas y los tornillos. La lista puede incluir información útil, como las dimensiones de una pieza y un número de pieza para que pueda solicitar una pieza nueva.

Los proyectos de código abierto ahora incluyen una *Lista de materiales de software* (SBOM, pronunciada “S-bomb”). Como mínimo, un SBOM enumera paquetes, nombres de archivos, licencias, autores o propietarios y números de versión. Muchos SBOM van más allá e incluyen información sobre vulnerabilidades de seguridad.

Los proyectos generan un SBOM para cada versión utilizando herramientas automatizadas. Los usuarios pueden escanear el SBOM para encontrar la información que necesitan. Por ejemplo, extraer licencias ayuda a la organización a decidir rápidamente si los componentes son compatibles con su propio código y si está legalmente permitido utilizar los componentes con su código.

De manera similar, extraer información de la versión de cada paquete ayuda a la organización a descubrir si diferentes partes de su producto dependen de diferentes versiones de una biblioteca en particular. El uso de dos versiones de la biblioteca provoca, como mínimo, hinchazón. También podría crear confusión y errores si un componente se construye con la versión incorrecta.

Estándares para SBOM

Debido a que los entornos informáticos utilizan enormes cantidades (a veces decenas de miles) de componentes y realizan cambios frecuentes, los SBOM deben estar altamente estructurados para que la información se pueda extraer de forma automática y rápida. Esta sección describe dos de los formatos más populares en el mundo del código abierto:

Intercambio de datos de paquetes de software (SPDX)

El formato representa cada paquete y todo el contenido de ese paquete en una estructura de árbol. El formato documenta las dependencias entre archivos y otras relaciones. Se pueden crear punteros a información, conocidos como “fragmentos” o “snippets”, y luego utilizarlos en todo el documento, de modo que la información deba definirse en un solo lugar. Este formato

fue desarrollado por la Fundación Linux.

CycloneDX

Este es un formato más grande con campos más detallados, particularmente para información de vulnerabilidad. El formato fue creado por el *Open Worldwide Application Security Project* (OWASP) y es popular en el ejército y otras organizaciones centradas en la seguridad. Por ejemplo, una entrada de un archivo podría incluir el nombre de una vulnerabilidad de seguridad, la fuente de información sobre la vulnerabilidad, los objetivos afectados y más. Este formato también está diseñado para implementaciones en la nube que pueden incluir miles de sistemas diferentes.

Tanto SPDX como CycloneDX crean estructuras jerárquicas en varios formatos, incluidos JSON y XML. Existen muchas herramientas para cada estándar, tanto para crear los formatos como para escanear los archivos creados en busca de información. Los sitios populares de repositorios de control de versiones permiten a los desarrolladores crear un SBOM en uno de estos formatos con solo hacer clic en un botón.

Análisis de composición de software

Saber qué está utilizando una organización requiere una evaluación de su software que incluya una comprensión de su procedencia, qué vulnerabilidades contiene, qué licencias utiliza y otros factores que ayudan a un individuo u organización a decidir si utilizar el software. Esta tarea se llama *Análisis de composición de software* (SCA) y existen muchas herramientas para realizar análisis sofisticados de SBOM y del propio software.

Algunas herramientas extraen información de licencia, lo que ayuda a una organización a decidir si es seguro incluir el software en un producto y si los diferentes componentes tienen licencias compatibles. Algunas herramientas incluso comparan fragmentos de código con bibliotecas de proyectos de código abierto comunes para descubrir si el código se tomó de los proyectos sin la atribución adecuada.

El uso de estas herramientas es especialmente crucial durante una fusión o adquisición. La empresa adquirente podría descubrir que el software que desea adquirir es menos valioso de lo que esperaba, porque contiene componentes de código abierto que imponen requisitos que interfieren con su uso previsto en la nueva organización.

Políticas formales y cumplimiento

Los procesos y técnicas descritos en esta lección deben ser diseñados por gerentes con el aporte de desarrolladores, abogados y otras personas con conocimientos. Esta sección describe algunas de las decisiones políticas que las organizaciones deben tomar con respecto al software de código

abierto. Terminaremos con una descripción de una *Oficina de Programas de Código Abierto* (OSPO), que puede ser un defensor y una fuente de información para las políticas.

Principios que rigen el uso y la contribución del software de código abierto

Las organizaciones pueden beneficiarse enormemente del uso y la contribución al software de código abierto, pero deben hacerlo de manera que protejan sus propios intereses y beneficien a los proyectos de código abierto. Se deben definir políticas para:

- Dónde utilizar software de código abierto (operaciones, proyectos internos, productos de cara al cliente, etc.)
- Qué hace que un proyecto de código abierto sea apropiado para la organización: características, rendimiento, seguridad, hoja de ruta para futuras extensiones y viabilidad de la comunidad.
- Escaneos para análisis de composición de software y dónde se debe almacenar la documentación resultante.
- Recompensas para los desarrolladores que contribuyen al software de código abierto, incluida su participación en foros públicos
- Lineamientos para contribuir a proyectos, incluyendo cómo ser representante público de la empresa.
- Requisitos de documentación, para que los desarrolladores fuera del equipo central puedan entender cómo contribuir e interactuar.

La OSPO puede coordinar la creación de estas políticas y la educación para garantizar su cumplimiento.

Acuerdos de colaborador

Los proyectos de código abierto deben garantizar que las contribuciones sean legítimas: por ejemplo, que el código no haya sido tomado de algún producto propietario o de un proyecto de código abierto con una licencia incompatible. Muchos proyectos de código abierto requieren que los desarrolladores proporcionen un documento llamado *Acuerdo de licencia de colaborador* (CLA) para garantizar la legitimidad de su contribución.

Los desarrolladores de una organización que contribuyen a estos proyectos pueden solicitar a los abogados de la organización que revisen y aprueben el CLA. Por lo tanto, los abogados deben comprender la relevancia de las cláusulas de los CLA y estar preparados para examinarlas.

Se han emitido numerosas críticas sobre los CLA, incluida su complejidad y las brechas que dejan

para que los proyectos utilicen el código aportado de formas no deseadas por los contribuyentes.

Muchos proyectos, en lugar de un CLA, solicitan al desarrollador que firme un breve documento llamado *Certificado de origen del desarrollador* (DCO), que certifica que el desarrollador tiene derecho a contribuir con el código. La DCO, que se analiza en otra lección, generalmente no se presenta a un abogado para su revisión.

Algunos proyectos simplemente piden a los contribuyentes que firmen los derechos de autor de su código en el proyecto en un *Acuerdo de cesión de derechos de autor* (CAA). Sin embargo, a muchos contribuyentes no les gusta esta práctica porque no pueden usar el código en algún proyecto propio y porque otorga mucho poder al proyecto.

La Oficina del Programa de Código Abierto (OSPO)

Los proyectos de código abierto combinan factores sociales, técnicos, legales y organizativos inusuales. En este momento, el conocimiento de estos factores no se ha extendido por las organizaciones hasta el punto de que todos los gerentes, desarrolladores, abogados y otros los comprendan profundamente. Por lo tanto, las organizaciones pueden beneficiarse enormemente de la creación de una *Oficina de Programas de Código Abierto* (OSPO) como punto central para la promoción, la formulación de políticas, la aplicación de políticas y la educación.

Como una especie de departamento multipropósito, los OSPO pueden desempeñar muchas funciones, tales como:

Promoción del código abierto

Las OSPO pueden difundir tanto los conceptos detrás del movimiento del código abierto como sugerencias para el uso del código abierto en toda su organización. Pueden recordar a los desarrolladores que busquen soluciones de código abierto para los problemas que están resolviendo y animarlos a adoptar el software adecuado. Pueden instar a los gerentes a que concedan tiempo a los desarrolladores para participar en proyectos de código abierto y a reconocer esa participación al evaluar a los desarrolladores para aumentos salariales y ascensos laborales.

Definición de políticas

Las OSPO pueden movilizar a los administradores para crear políticas y establecer repositorios para ellas.

Aplicación de políticas

Las OSPO pueden recordar a los desarrolladores que escaneen el software y los SBOM y que sigan las reglas corporativas sobre el uso del código abierto. Las OSPO pueden conservar documentación sobre el software adoptado y otros aspectos del uso corporativo.

Educación

Los OSPO pueden explicar a los desarrolladores cómo evaluar y participar en proyectos de código abierto, explicar a los abogados los detalles de las licencias y otras consideraciones legales, y ayudar a la empresa a comprender los cambios organizacionales y culturales que facilitan el uso del software de código abierto.

Existen numerosos documentos y recursos en línea para crear una OSPO. Una organización pequeña puede encargar la tarea a un tercero que sea experto en esa área.

Ejercicios guiados

1. ¿Por qué un desarrollador no debería simplemente tomar el código que le gusta de un proyecto de código abierto y ponerlo en su propio código sin conservar la licencia?

2. ¿Qué tipo de documentos firmaría un desarrollador al realizar una contribución a un proyecto de código abierto?

3. Encuentra algún software de código abierto que sería una base excelente para su sitio minorista. ¿Puede superponer su logotipo sobre su logotipo popular para crear una imagen llamativa para su sitio web?

Ejercicios exploratorios

1. ¿Qué consideraciones podrían llevarlo a crear una bifurcación de un proyecto de código abierto para su propio uso, a pesar de los inconvenientes de las bifurcaciones?

2. Cuando encuentra algún software de código abierto que satisface sus necesidades, ¿cuáles son algunas de las razones por las que podría rechazarlo?

3. Le gustaría utilizar una herramienta de registro con copyleft en su producto propietario. ¿Existe alguna manera de combinarlos que no requiera que ofrezcas tu producto bajo la licencia copyleft?

Resumen

Esta lección ha cubierto muchas consideraciones a tener en cuenta antes de utilizar software gratuito y de código abierto. Explicó cómo cumplir con las licencias, diversos riesgos legales, de reputación y financieros, y cómo definir políticas importantes y hacerlas cumplir dentro de su organización.

Respuestas a ejercicios guiados

1. ¿Por qué un desarrollador no debería simplemente tomar el código que le gusta de un proyecto de código abierto y ponerlo en su propio código sin conservar la licencia?

   Esto suele ser una violación de la licencia de código abierto y también representa plagio e infracción de derechos de autor. En la práctica, la organización puede verse obligada a cumplir con la licencia, con consecuencias que van desde la vergüenza y el daño a la reputación hasta la destrucción de su modelo de negocio si se mezcla código copyleft con código propietario.

2. ¿Qué tipo de documentos firmaría un desarrollador al realizar una contribución a un proyecto de código abierto?

   Un Acuerdo de licencia de colaborador es un documento legal que otorga derechos a la organización de código abierto para usar el código. Un Certificado de origen de desarrollador es un documento más breve y sencillo que promete que el desarrollador tiene derecho a contribuir con el código. Un Acuerdo de cesión de derechos de autor otorga todos los derechos a la organización receptora.

3. Encuentra algún software que sería una base excelente para su sitio minorista. ¿Puede superponer su logotipo sobre su logotipo popular para crear una imagen llamativa para su sitio web?

   Si el proyecto tiene su logotipo como marca registrada, su cambio probablemente violaría la marca registrada. Incluso si el logotipo no es una marca registrada, su cambio podría considerarse irrespetuoso y confuso.

Respuestas a ejercicios exploratorios

1. ¿Qué consideraciones podrían llevarlo a crear una bifurcación de un proyecto de código abierto para su propio uso, a pesar de los inconvenientes de las bifurcaciones?

   Si está satisfecho con el estado actual del código, es posible que no necesite mantenerse al día con los cambios en el proyecto principal. Es posible que desee crear un producto sustancialmente diferente de los usos que se le da al proyecto principal y, por lo tanto, esté dispuesto a separarse del proyecto principal. El código puede ser lo suficientemente valioso en su plan de negocios como para que su equipo esté dispuesto a asumir la responsabilidad total de su desarrollo y mantenimiento.

2. Cuando encuentra algún software de código abierto que satisface sus necesidades, ¿cuáles son algunas de las razones por las que podría rechazarlo?

   El código puede contener demasiados errores y vulnerabilidades de seguridad, es posible que la comunidad de desarrolladores del proyecto no funcione bien, es posible que no le guste la dirección en la que evoluciona el código o que la licencia no sea compatible con otro código de su producto.

3. Le gustaría utilizar una herramienta de registro con copyleft en su producto propietario. ¿Existe alguna manera de combinarlos que no requiera que ofrezcas tu producto bajo la licencia copyleft?

   Integrar el código de la herramienta en su código podría activar el requisito recíproco del copyleft, dependiendo de qué licencia esté vigente. La herramienta de registro debe estar separada de su producto para que su producto no se vea como un derivado. Por ejemplo, podría ser seguro ejecutar la herramienta copyleft como un proceso separado y comunicarse con ella mediante el paso de mensajes.

Linux Professional Institute

Tema 055: Gestión de proyectos

055.1 Modelos de desarrollo de softwar

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 055.1

Peso

3

Áreas de conocimiento clave

- Comprender la relevancia y los objetivos de la gestión de proyectos en el desarrollo de software
- Comprensión básica del desarrollo de software en cascada
- Comprensión básica del desarrollo ágil de software, incluidos Scrum y Kanban
- Comprender el concepto de DevOps

Lista parcial de archivos, términos y utilidades

- Fases en proyectos en cascada (ingeniería de requisitos, análisis de negocio, diseño de software, desarrollo, pruebas, operaciones)
- Roles en proyectos en cascada (gerentes de proyectos, analistas de negocios, arquitectos de software, desarrolladores, evaluadores)
- Organización de proyectos Scrum (sprints y planificación de sprints, backlog de productos y sprints, scrums diarios, revisión de sprints y retrospectiva de sprints)
- Roles en proyectos Scrum (propietarios de productos, desarrolladores, scrum masters)
- Organización de proyectos Kanban (tableros Kanban)

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	055 Gestión de Proyectos
Objectivo:	055.1 Modelos de desarrollo de software
Lección:	1 de 1

Introducción

La vida está llena de proyectos. Un proyecto podría ser planificar una noche de cine, un viaje o un evento familiar, programar la semana de los niños o mejorar su trabajo en un pasatiempo. No importa lo que quieras llevar a cabo: necesitas hacer planes y tomar acciones. Normalmente creamos varios escenarios, acompañados de un plan B, plan C, etc. Esto no es diferente en el mundo del desarrollo de software.

Roles en el desarrollo de software

Se necesita una variedad de habilidades para una gestión exitosa de proyectos, por lo que es útil encontrar diferentes personas para desempeñar roles bien definidos. Las siguientes secciones describen algunas de las funciones que se encuentran comúnmente en proyectos de software.

Gerente de Proyecto

Gestionar un proyecto es una tarea muy compleja. Algunas situaciones parecen fáciles al principio, pero se necesita cuidado y experiencia para resolverlas correctamente. La persona que hace esto es conocido como el *gerente de proyecto*.

Las responsabilidades del gerente de proyecto incluyen garantizar que el proyecto se termine a tiempo, dentro del presupuesto y con una calidad aceptable. Incluso si no escriben una sola línea de código, los gerentes de proyecto son los que son responsables y rinden cuentas de los resultados del equipo. Tienen que alinearse con los clientes en cuanto a los plazos. También hablan con las personas que desempeñan otros roles para asegurarse de que todo esté en orden.

Analista de negocios e ingeniero de requisitos

No hay lugar para la microgestión, al menos no en un lugar de trabajo saludable. Los *analistas de negocios* (BA) y los *ingenieros de requisitos* (RE), internos o externos al equipo del proyecto, son la mano derecha de los directores de proyecto. Son responsables de crear documentación concisa y clara sobre los requisitos. También son el puente entre los clientes y los desarrolladores. Los clientes se comunican a través de un lenguaje sencillo, que BA/RE garantiza que proporcionará documentación clara e inequívoca con la que los desarrolladores podrán realizar su trabajo de forma eficaz.

Imagínese que le piden que traiga un “gran pastel” a una fiesta. ¿Qué significa “grande”? ¿Diez rebanadas o veinte? Quizás sea el pastel de una gran boda y deberían ser cien porciones. El BA/RE tiene que especificar requisitos tales como cantidades exactamente, de modo que los desarrolladores sepan, por ejemplo, que una aplicación está diseñada para un número específico de usuarios.

Estamos apenas comenzando a definir los requisitos, pero ya se puede ver la importancia de una comunicación clara. La comunicación es esencial para cualquier trabajo conjunto, pero quizás sea incluso más importante en un proyecto de código abierto, ya que personas con antecedentes muy diferentes pueden trabajar en cada parte del proyecto. Alguien puede colaborar como estudiante, alguien más como padre, una persona jubilada, entre otros. Aun así, el progreso tiene que ser fluido en este entorno, por lo que la comunicación no puede retrasar el progreso.

Desarrolladores, arquitectos y evaluadores

Para implementar los requisitos, un proyecto de software necesita *desarrolladores* que creen el producto en función de la documentación y las decisiones de diseño. Su trabajo es juzgado por el *arquitecto*, quien generalmente tiene el conocimiento técnico y de dominio más extenso que cualquiera en el proyecto. Cada desarrollador, especialmente los de alto nivel, es responsable de su propio código, pero las decisiones importantes las toma o aprueba el arquitecto.

A veces, las personas son designadas específicamente como *probadores*, quienes son responsables de todo tipo de pruebas, documentando los resultados y creando tickets de problemas.

Planificación y programación

Ahora que hemos discutido los roles básicos, hablemos de las fases de un proyecto: planificación, programación, implementación, mantenimiento/prueba y entrega. Algunas fases difieren en varios modelos de desarrollo de software, pero la planificación y la programación son temas generales que discutiremos antes de profundizar en los diferentes modelos.

La *planificación* se lleva a cabo después de que los BA/RE proporcionen la lista de requisitos. La planificación consiste en una o más reuniones sobre lo que el equipo quiere lograr, cómo planean lograrlo y cuánto tiempo llevaría. Por lo general, en estas reuniones también se realizan algunos análisis de riesgos.

Luego, los planificadores tienen que *programar* el trabajo y crear *hitos*. En cada hito, el equipo sabe que se ha hecho una parte grande e importante. Los componentes o funciones a largo plazo pueden tardar meses, por lo que los planificadores deben tener en cuenta las vacaciones, los feriados y, especialmente durante las estaciones frías, algunas bajas por enfermedad, para garantizar un tiempo de entrega preciso y evitar prisas y pánico al acercarse la fecha límite. Con un buen cronograma, los desarrolladores se sienten más relajados y pueden entregar soluciones de alta calidad a tiempo, sin averías.

Herramientas comunes

Dos herramientas que benefician a todos los modelos, y a la gestión de proyectos en general, son un *sistema de control de versiones* (VCS) y una plataforma de *gestión del ciclo de vida de las aplicaciones* (ALM). El control de versiones se analiza en profundidad en otra lección, así que digamos algunas palabras sobre las ALM.

ALM es un marco integral que gestiona el ciclo de vida de una aplicación de software desde el desarrollo inicial hasta el mantenimiento y el eventual retiro. ALM integra personas, procesos y herramientas para mejorar la colaboración y la eficiencia, garantizando que todas las etapas del ciclo de vida del software estén bien coordinadas y alineadas con los objetivos comerciales. Este enfoque ayuda a mantener la calidad, el rendimiento y la confiabilidad de las aplicaciones durante toda su vida útil.

Ahora que tenemos una descripción general de los roles y las fases del ciclo de vida, profundicemos en los modelos.

Modelo de cascada

Esta es una de las metodologías más tempranas y sencillas en el desarrollo de software. Puedes imaginarlo como una escalera, donde es necesario terminar cada escalón antes de seguir

adelante. Pero el modelo va en una sola dirección, lo que lo hace adecuado para proyectos con requisitos claros y cambios mínimos o nulos esperados.

Aunque la simplicidad de este modelo puede ser una ventaja, la metodología rígida puede ser un inconveniente cuando se necesitan flexibilidad y adaptabilidad. Y hoy en día, normalmente todo cambia mientras trabajas en tu proyecto.

Como se explicó en las secciones anteriores, para comenzar el desarrollo, un proyecto debe tener los requisitos, un plan terminado y un cronograma de trabajo finalizado con hitos. En el modelo en cascada, estos son el resultado de la ingeniería de requisitos y el análisis empresarial.

Ingeniería de requisitos

En esta fase inicial, se recopilan y documentan todos los requisitos del proyecto. Este trabajo implica extensas discusiones entre el director del proyecto y las partes interesadas para comprender sus necesidades y expectativas. El resultado es un documento completo de especificación de requisitos que describe lo que debe hacer el sistema.

Análisis de Negocios

Los analistas de negocios examinan los requisitos desde una perspectiva comercial para asegurarse de que se alineen con los objetivos de la organización. Los BA realizan estudios de viabilidad, evaluaciones de riesgos y análisis de costo-beneficio para determinar si el proyecto es viable y vale la pena. Los conocimientos adquiridos se utilizan para perfeccionar el alcance y los objetivos del proyecto.

Diseño de software

El diseño de software es el paso en el que el arquitecto crea una especificación de diseño detallada para que la sigan los desarrolladores. Dada esta especificación, el equipo puede comenzar a implementar los requisitos; en otras palabras, iniciar la fase de desarrollo, que viene a continuación.

Desarrollo

La fase de desarrollo es donde ocurre la magia: donde se escribe el código. Siguiendo asiduamente la documentación y las decisiones de diseño, los desarrolladores escriben sus partes. Después de que los desarrolladores revisen su código con otros desarrolladores o con el arquitecto, esta fase se puede considerar finalizada.

Pruebas

Al desarrollo le sigue una fase de pruebas, gestionada por los evaluadores. Hay diferentes tipos de pruebas, que se describen en otra lección, como pruebas unitarias, de integración, sistemas y pruebas de aceptación.

El objetivo de estas pruebas es sacar a la superficie los problemas antes del lanzamiento y permitir que los desarrolladores los solucionen a tiempo, no después de que el proyecto se haya implementado y publicado y los usuarios puedan sentirse irritados y frustrados por los errores.

Operaciones

Una vez superadas las pruebas y corregidos los errores, el proyecto puede publicarse, es decir, implementarse en el entorno de producción. Esta fase puede contener instalación, configuración y, a veces, incluso la capacitación del usuario. Una vez que el proyecto está listo para usarse, se encuentra en una fase de mantenimiento, donde el equipo puede manejar los problemas de los usuarios y entregar actualizaciones si es necesario.

Evaluación del modelo de cascada

¿Qué has notado al leer sobre este modelo? ¿Lo encuentras eficiente? ¿Falta algo? Veamos los pros y los contras.

El modelo en cascada se beneficia de:

Estructura clara

El proceso lineal hace que sea fácil de gestionar, comprender y seguir.

Documentación meticulosa

la documentación detallada y exhaustiva en cada fase ayuda al equipo a comprender lo que se necesita durante el desarrollo y el mantenimiento futuro.

Previsibilidad

Los hitos claros y definidos ayudan a proporcionar un cronograma y un presupuesto previsibles.

Gestión de proyectos más sencilla

Gracias a su flujo lineal y sencillo, el modelo es fácil de gestionar.

El modelo de cascada sufre de:

Rigidez

La inflexibilidad lineal del modelo dificulta la implementación de cambios una vez finalizada la fase de requisitos.

Pruebas tardías

Durante la fase de desarrollo, las pruebas son desordenadas o faltan, lo que puede llevar a un reconocimiento tardío de problemas importantes.

Supuesto de requisitos perfectos

El modelo requiere y supone que cada requisito sea correcto y claro, lo que puede resultar muy poco realista. El modelo no deja espacio para realizar comprobaciones durante el desarrollo para garantizar que los requisitos sean precisos y comprendidos por los desarrolladores.

Falta de feedback

Sólo en la última fase un cliente puede decir algo sobre el producto. Entonces, si algo (o muchas cosas) no cumple con sus expectativas, el problema aparece sólo al final.

Desarrollo de software ágil, Scrum y Kanban

Para explorar este conjunto de modelos de desarrollo de software más modernos, comencemos con la palabra *ágil*: ¿Qué significa? La agilidad expresa la capacidad de reaccionar y moverse rápidamente, tanto en el mundo físico como en el mental. El objetivo es el mismo en el desarrollo de software.

El *desarrollo ágil de software* es una metodología que se basa en el desarrollo iterativo, la flexibilidad y la colaboración. Se centra en ofrecer pequeñas mejoras y al mismo tiempo recopilar comentarios e implementarlos durante el desarrollo. Este enfoque permite reacciones, ajustes y respuestas rápidas, ahorrando tiempo y asegurando que el proyecto cumpla con las expectativas de los usuarios y clientes.

El movimiento ágil fue lanzado en 2001 mediante un *Manifiesto para el desarrollo de software ágil* en línea que enumeraba sus principios de la siguiente manera:

> Estamos descubriendo mejores formas de desarrollar software haciéndolo y ayudando a otros a hacerlo.
>
> A través de este trabajo hemos llegado a valorar:
>
> - *Individuos e interacciones* sobre procesos y herramientas
> - *Software funcional* sobre documentación integral

> - *Colaboración con el cliente* sobre negociación de contratos
> - *Respuesta al cambio* sobre seguir un plan
>
> Es decir, si bien hay valor en los elementos de la derecha, valoramos más los elementos de la izquierda.
>
> — Manifiesto para el desarrollo de software ágil

Scrum

Scrum es uno de los marcos más populares (quizás el más popular) dentro del desarrollo ágil de software. Scrum se basa en los siguientes componentes básicos:

> En pocas palabras, Scrum requiere un Scrum Master para fomentar un entorno donde:
>
> 1. Un Product Owner ordena el trabajo de un problema complejo en un Product Backlog.
> 2. El Scrum Team convierte una selección del trabajo en un Incremento de valor durante un Sprint.
> 3. El Scrum Team y sus stakeholders inspeccionan los resultados y los ajustan para el próximo Sprint.
> 4. Repetir
>
> — La Guía Scrum 2020

¿Pero quién es el scrum master y el propietario del producto? ¿Qué es un product backlog y un sprint? Profundicemos en los roles y procesos.

Sprints y planificación de sprints

Un *sprint* es una fase de desarrollo que normalmente dura de dos a cuatro semanas, durante la cual se completan algunas tareas y/o características previamente acordadas. Para llegar a un acuerdo sobre las tareas, entra la *planificación de sprint*. Este proceso es donde el equipo toma decisiones sobre qué tareas se pueden completar durante el próximo sprint.

Estas elecciones se basan en las prioridades establecidas por el *propietario del producto* (PO), que controla el proyecto y, a menudo, proporciona su financiación.

Product Backlog y Sprint Backlog

Como se acaba de explicar, el PO gestiona la lista de prioridades, que contiene todo el trabajo deseado en el proyecto. En Scrum, esta lista se llama *product backlog*. Cada *sprint backlog* contiene

las tareas seleccionadas para el sprint actual del *product backlog*. El sprint backlog contiene los elementos elegidos por el equipo durante la planificación del sprint.

Daily Scrum or Stand-up

Las reuniones *diarias Scrum* o *stand-up* son reuniones breves y eficientes en las que los equipos discuten su progreso, resaltan los problemas y los obstáculos y comparten sus planes para el día. Por lo general, se realiza al comienzo de la jornada laboral.

Incremento

Un *incremento* es un objetivo alcanzado durante un sprint. Cada sprint debe resultar en uno o más incrementos. La corrección de la tarea o características logradas por el incremento debe verificarse mediante pruebas.

Revisión del sprint

La *revisión del sprint* es una reunión para inspeccionar el resultado del sprint. Estas reuniones suelen ser más que demostraciones: el objetivo es obtener comentarios basados en los resultados del equipo Scrum. Como el objetivo es alcanzar el objetivo del producto, las partes interesadas dan opiniones y sugerencias sobre futuras adaptaciones. El resultado de estas reuniones pueden ser cambios o adiciones a la cartera de productos.

Sprint Retrospective

El objetivo del *Sprint Retrospective* es mejorar la calidad y la eficacia, y no sólo en el producto actual. La retrospectiva debe revelar tensiones ocultas dentro del equipo, cualquier falta de información que ralentice los procesos, dependencias externas que deberían aliviarse para aligerar la carga del equipo, etc. El equipo discute qué salió bien, qué problemas enfrentaron y qué soluciones se encontraron (o no) para esos problemas.

El resultado es una lista de problemas junto con posibles soluciones asignadas a la persona responsable.

Scrum Master

Todas estas reuniones están organizadas por el *Scrum master*. Cada equipo Scrum necesita uno. Son responsables de facilitar los procesos de Scrum y ayudar al equipo a seguir avanzando hacia el objetivo del producto mientras se encuentran problemas durante los sprints. El Scrum master no sólo guía los procesos, sino que actúa como coach para asegurar una comunicación efectiva dentro del equipo.

Product Owner

El *propietario del producto* es responsable de maximizar el valor del producto creado por el equipo Scrum. Este rol puede variar entre diferentes organizaciones y equipos. Incluso al delegar tareas, el propietario del producto sigue siendo responsable de los resultados.

El éxito requiere respeto organizacional por las decisiones del propietario del producto, que se reflejan en el trabajo pendiente del producto y el incremento de la revisión del sprint. El propietario del producto es una sola persona que representa las necesidades de varias partes interesadas en el trabajo pendiente del producto. Cualquiera que desee cambiar el trabajo pendiente debe persuadir al propietario del producto para que lo haga. El propietario del producto define y prioriza la visión y el trabajo pendiente del producto y garantiza la transparencia, visibilidad y comprensibilidad del trabajo pendiente.

El Scrum Master y el propietario del producto colaboran para impulsar el éxito del proyecto. Su asociación ayuda al equipo de desarrollo a ofrecer funciones de alto valor de manera eficiente y eficaz.

Desarrolladores

Los desarrolladores implementan los requisitos siguiendo el sprint backlog acordado. Pueden trabajar de forma independiente, pero hay varias ocasiones en las que pueden colaborar: por ejemplo, programando en pareja o revisando el código de otro desarrollador.

Evaluación del modelo Scrum

Scrum se ha vuelto el favorito entre los equipos de software, pero también tiene pros y contras.

El Modelo Scrum se beneficia de:

Flexibilidad
: Los procesos son flexibles e iterativos, lo que permite cambios basados en la retroalimentación.

Participación del cliente
: La retroalimentación periódica de las partes interesadas garantiza la alineación con las necesidades del cliente.

Calidad mejorada
: Las pruebas y revisiones frecuentes mejoran la calidad del producto.

Colaboración en equipo
: El modelo enfatiza el trabajo en equipo y la comunicación a través de reuniones diarias y

periódicas.

El modelo Scrum sufre de:

Disciplina requerida

Scrum exige un estricto cumplimiento de sus prácticas, lo que puede resultar un desafío.

Aumento del alcance

Existe el riesgo de que se agreguen cada vez más solicitudes de clientes y retrasen el lanzamiento, si la acumulación de productos no se gestiona bien.

Confusión de roles

Los roles estándar, como Scrum Master y Product Owner, pueden malinterpretarse o implementarse mal.

Intensidad de recursos

Las reuniones diarias y las revisiones frecuentes pueden consumir mucho tiempo.

Kanban

Kanban es otro marco ágil popular que enfatiza la visualización del trabajo, la gestión del flujo y las mejoras de procesos. Una de las grandes diferencias entre Scrum y Kanban es que Kanban no requiere iteraciones de longitud fija. Por lo tanto, hace que la entrega continua sea más flexible y puede equilibrar diferentes prioridades de trabajo.

Veremos qué necesitan los equipos para un proyecto Kanban en las siguientes subsecciones.

Tableros Kanban

Un tablero Kanban es una herramienta visual que rastrea el flujo del trabajo a través de etapas. Las columnas principales y más importantes son “To Do”, “In Progress” y “Done”. Se pueden agregar más columnas, pero estas tres columnas principales deben estar en el tablero. Esta visualización ayuda a los equipos a detectar cuellos de botella y ver el estado de las tareas en un abrir y cerrar de ojos.

Límite de trabajo en curso

Un límite de trabajo en progreso (WIP por sus siglas inglés) ayuda a evitar la multitarea y mejora la concentración. Con este límite, llega un punto en el que no se pueden agregar más tareas a la columna “In Progress”. De esta forma, los desarrolladores no se verán sobrecargados de tareas y podrán centrarse en las tareas en las que están trabajando actualmente.

Miembros del equipo

Los miembros del equipo son responsables de completar las tareas y establecer un flujo a través del sistema Kanban. Colaboran para priorizar el trabajo, resaltar y abordar cualquier tipo de problema que surja.

Administrador Kanban

El *administrador Kanban* supervisa el proceso Kanban y garantiza que el equipo siga los principios y prácticas de Kanban. Este administrador facilita reuniones, monitorea los flujos de trabajo y ayuda a resolver cualquier problema que pueda surgir.

Evaluación del modelo Kanban

El núcleo de este modelo es simple. Veamos los pros y los contras.

El Modelo Kanban se beneficia de:

Flujo de trabajo visual
: los tableros Kanban brindan una visibilidad clara del estado y el progreso del trabajo.

Flexibilidad
: Al carecer de iteraciones fijas, el modelo admite la entrega continua con una adaptabilidad sustancial.

Límites de tareas
: El límite de WIP ayuda a evitar que los miembros del equipo se sobrecarguen y, por lo tanto, mejora su concentración y eficiencia.

Mejora continua
: El modelo fomenta la evaluación periódica y la mejora de los procesos.

El modelo Kanban sufre de:

Falta de plazos
: Sin plazos establecidos, la gestión del tiempo es un desafío.

Menos estructura
: El modelo puede carecer de la estructura que algunos equipos necesitan para mantenerse organizados.

Disciplina requerida

Los límites efectivos de WIP y la gestión de la junta requieren una atención cuidadosa.

Posible simplificación excesiva

Las descripciones de las tareas pueden simplificar demasiado proyectos complejos si no se implementan cuidadosamente.

DevOps

DevOps es un enfoque de desarrollo de software que integra los equipos de desarrollo (Dev) y operaciones (Ops) para mejorar la colaboración, la eficiencia y la velocidad de entrega. El objetivo principal de DevOps es acortar el ciclo de vida del desarrollo de software y proporcionar una entrega continua con alta calidad de software. DevOps enfatiza la automatización, la integración continua y la entrega continua (CI/CD) y la estrecha colaboración entre equipos tradicionalmente aislados.

La automatización es crucial en DevOps para tareas como la integración de código, pruebas, implementación y gestión de infraestructura. La automatización de tareas repetitivas reduce los errores, acelera los procesos y permite a los equipos centrarse en un trabajo más estratégico.

El monitoreo y el registro continuos son esenciales para mantener la salud y el rendimiento del sistema. Las prácticas de DevOps incluyen la configuración de sistemas integrales de monitoreo y registro para detectar problemas, analizar tendencias y mejorar la confiabilidad del sistema.

Evaluación del modelo DevOps

Aunque DevOps es altamente recomendado para muchos tipos de proyectos de software modernos, aún se deben considerar los pros y los contras.

El modelo DevOps se beneficia de:

Velocidad y eficiencia

El modelo acelera los ciclos de desarrollo e implementación a través de la automatización.

Colaboración mejorada

El modelo cierra la brecha entre los equipos de desarrollo y operaciones.

Entrega continua

El modelo garantiza que el software esté siempre en un estado implementable, lo que lleva a lanzamientos más frecuentes.

Alta confiabilidad

Las pruebas y el monitoreo automatizados mejoran la confiabilidad y reducen los errores.

El modelo DevOps sufre de:

Cambio cultural

El modelo requiere un cambio cultural significativo y aceptación en toda la organización.

Complejidad

La gestión de canales de CI/CD y la infraestructura automatizada puede ser compleja.

Riesgos de seguridad

La implementación continua puede introducir vulnerabilidades de seguridad si no se gestiona con cuidado.

Sobrecarga de herramientas

DevOps puede resultar abrumador debido a la multitud de herramientas y tecnologías involucradas.

Ejercicios guiados

1. ¿Qué significa agilidad?

2. ¿Cuál es la definición de Scrum, según la Guía Scrum?

3. ¿Por qué Scrum puede funcionar mejor que el modelo en cascada con respecto a los comentarios de los clientes?

4. ¿Cuáles son los aspectos positivos de DevOps?

Ejercicios exploratorios

1. ¿Por qué el modelo en cascada no es el mejor para usar en un entorno que cambia rápidamente?

2. ¿Qué puede pasar cuando el rol del Scrum master está mal implementado?

Resumen

La relevancia de la gestión de proyectos en el desarrollo de software, especialmente en proyectos de código abierto, radica en su capacidad para proporcionar estructura, mejorar la comunicación, definir roles, gestionar riesgos y garantizar la calidad. Estos elementos son cruciales para la finalización exitosa de los proyectos y para fomentar un entorno de desarrollo colaborativo y productivo.

En esta lección, analizamos el modelo en cascada, las metodologías ágiles y DevOps. El conocimiento de estos modelos es esencial para comprender la gestión de proyectos en el desarrollo de software. No existe una forma perfecta de trabajar: cada proyecto es diferente. En esta lección, aprendió las funciones, los procesos y los pros y los contras de varios enfoques, que pueden ayudarlo en el futuro a comprender y quizás elegir el modelo correcto para un proyecto.

Respuestas a ejercicios guiados

1. ¿Qué significa agilidad?

   La agilidad expresa la capacidad de reaccionar y moverse rápidamente, tanto en el mundo físico como mental.

2. ¿Cuál es la definición de Scrum, según la Guía Scrum?
   - Un Product Owner ordena el trabajo para un problema complejo en un Product Backlog.
   - El Scrum Team convierte una selección del trabajo en un Incremento de valor durante un Sprint.
   - El Scrum Team y sus stakeholders inspeccionan los resultados y los ajustan para el próximo Sprint.
   - Repetir
3. ¿Por qué Scrum puede funcionar mejor que el modelo en cascada con respecto a los comentarios de los clientes?

   Debido a que la retroalimentación se recopila durante el desarrollo, el producto final puede satisfacer mejor las expectativas del cliente.

4. ¿Cuáles son los aspectos positivos de DevOps?

   Velocidad y eficiencia — Colaboración mejorada — Entrega continua — Alta confiabilidad

== Respuestas a ejercicios exploratorios

5. ¿Por qué el modelo en cascada no es el mejor para usar en un entorno que cambia rápidamente?

   El modelo en cascada recopila comentarios solo al final del desarrollo. Por lo tanto, cualquier requisito que los desarrolladores no hayan entendido bien debe corregirse al final. Si el entorno cambia muy rápidamente, este modelo no debe usarse porque no hay provisión para retroalimentación en la mitad del ciclo de desarrollo.

6. ¿Qué puede pasar cuando el rol del Scrum master está mal implementado?

   Un rol de Scrum master mal implementado puede obstaculizar la capacidad del equipo para entregar productos de alta calidad de manera eficiente y puede socavar los beneficios de adoptar Scrum. Es posible que el equipo carezca de una orientación adecuada sobre las prácticas y principios de Scrum, lo que lleva a una aplicación inconsistente o incorrecta del marco.

   Sin un Scrum Master eficaz para eliminar los obstáculos, los impedimentos pueden ralentizar el progreso y reducir la productividad del equipo. Puede ocurrir una falta de comunicación entre los miembros del equipo y las partes interesadas, lo que resulta en malentendidos y expectativas desalineadas. El equipo puede experimentar una disminución de la moral y la motivación debido a problemas no resueltos, falta de apoyo y facilitación ineficaz de los eventos Scrum. Las ineficiencias pueden surgir de reuniones mal realizadas, falta de concentración y planificación y revisiones de sprint ineficaces.

055.2 Gestión de productos / Gestión de lanzamientos

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 055.2

Peso

2

Áreas de conocimiento clave

- Comprender los tipos de versiones comunes
- Comprender el control de versiones del software y los motivos de las versiones mayores o menores
- Comprender el ciclo de vida de un producto de software, desde su planificación, desarrollo y lanzamiento hasta su retiro
- Comprender la documentación de las versiones del producto

Lista parcial de archivos, términos y utilidades

- Versiones alfa y beta
- Liberar candidatos
- Congelación de funciones
- Lanzamientos mayores y menores
- Versionado semántico
- Hojas de ruta e hitos
- Registros de cambios
- Soporte a largo plazo (LTS)
- Fin de vida (EOL)

- Compatibilidad con versiones anteriores

Linux Professional Institute

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	055 Gestión de Proyectos
Objectivo:	055.2 Gestión de Productos / Gestión de Lanzamientos
Lección:	1 of 1

Introducción

La mayoría del software evoluciona de muchas maneras con el tiempo. De hecho, esa es una de las cosas maravillosas del software: es fácil de cambiar y solo requiere una transferencia de red para actualizar a todos los que usan el producto. Por eso, los desarrolladores agregan nuevas funciones, corrigen errores y fallas de seguridad, trasladan el software a nuevo hardware y agregan interfaces a programas y servicios populares. A medida que el software cambia, se lanzan nuevas *versiones* o *revisiones*.

Esta lección cubre la logística para planificar y realizar cambios en el software, cómo los equipos hacen malabarismos con múltiples versiones, convenciones para nombrar las versiones y otros aspectos organizacionales del desarrollo que son un subconjunto de las actividades involucradas en la *gestión de proyectos*. La logística que se aplica específicamente al momento, la denominación y el control de los lanzamientos se conoce como gestión de lanzamientos.

Características de las versiones (Releases)

Las versiones varían en términos de estabilidad, compatibilidad con versiones anteriores y

soporte. Examinaremos cada uno de esos conceptos en las siguientes secciones.

Versiones estables e inestables

La *estabilidad* es una característica clave que los usuarios esperan del software. La primera pregunta que muchos hacen al enterarse de una nueva versión es: "¿Qué tan estable es?". En otras palabras, ¿Funcionará de manera confiable o fallará, dañará los datos o producirá resultados incorrectos?

La estabilidad se puede medir en un espectro y muchas personas están felices de utilizar el software incluso cuando todavía hay algunos errores. Pero los equipos de desarrollo tienden a mantener las cosas simples y hablan de forma binaria sobre versiones *estable* e *inestable*.

La estabilidad se refiere tanto a los errores del software como a su probabilidad de cambiar de manera visible para los externos. Los desarrolladores podrían considerar inestable una versión de una biblioteca de software porque han cambiado las funciones que llaman los programadores (es decir, porque es posible que los programadores tengan que reescribir su software para la próxima versión). O el software puede ser inestable desde el punto de vista del usuario porque los desarrolladores han eliminado una función.

¿Hay alguna razón para lanzar una versión inestable? Sí: son valiosos para permitir a los usuarios probar nuevas funciones al principio del proyecto y detectar errores.

La mayoría de los proyectos lanzan revisiones muy tempranas de software para que las prueben los clientes clave; estas se denominan versiones *alfa*. Nadie debería utilizar estas versiones para un trabajo real; son sólo para pruebas. De hecho, los evaluadores deberían ejecutar las versiones alfa en computadoras que no estén realizando ningún trabajo importante, porque los errores en esas versiones podrían dañar los datos almacenados en la computadora.

Cuando el software está cerca de considerarse estable, el proyecto generalmente lanza otra revisión de prueba llamada versión *beta*. Estos aún no deben usarse para trabajos de producción, sólo para pruebas.

Tanto para las versiones alfa como para las beta, los desarrolladores tienen un proceso para informar errores y realizar un seguimiento del progreso hacia la corrección de estos.

Algunos proyectos incluyen otra etapa entre las versiones beta y estable, cuando los desarrolladores han corregido todos los errores que pueden y piensan que el producto está listo. Llaman a la versión *candidato de lanzamiento* y se la muestran a clientes o partes interesadas clave para que estas partes interesadas puedan planificar cómo utilizar las nuevas funciones.

Finalmente, algunos proyectos incluyen características que no son del todo estables, porque los

usuarios importantes las han solicitado. Los desarrolladores tienen la intención de que los usuarios prueben estas funciones y digan si les gustan, antes de finalizar las interfaces e invertir el esfuerzo para estabilizar las funciones.

Compatibilidad con versiones anteriores

La mayoría de los proyectos buscan *compatibilidad con versiones anteriores*, lo que significa que intentan no eliminar características o funciones que existían en versiones anteriores. La compatibilidad puede existir en varios niveles. Por ejemplo, si los desarrolladores mantienen la compatibilidad con versiones anteriores de la interfaz de programación de aplicaciones (API), prometen que el código fuente antiguo puede seguir funcionando, pero es posible que sea necesario volver a compilarlo. Si los proveedores de hardware o los desarrolladores de sistemas operativos mantienen la compatibilidad con versiones anteriores de la interfaz binaria de la aplicación (ABI), prometen que, los archivos ejecutables antiguos pueden ejecutarse en nuevas versiones del hardware.

Más adelante veremos las formas en que los proyectos manejan la *falta* de compatibilidad con versiones anteriores, cuando deciden que la interfaz antigua realmente no funciona para las nuevas características o entornos que necesitan soportar.

Soporte y fin de vida (EOL)

Idealmente, el software mejoraría cada vez más a medida que los desarrolladores descubran y solucionen sus problemas. Pero con el tiempo, las funciones pueden dejar de funcionar debido a cambios en el entorno del software. Además, se descubren nuevos errores que antes estaban ocultos.

Las nuevas versiones suelen combinar seguridad y corrección de errores con nuevas funciones. Es posible que no desee las nuevas funciones (de hecho, es posible que prefiera alguna función antigua que los desarrolladores eliminaron para dar paso a otras nuevas), pero las personas generalmente actualizan a la nueva versión para obtener las correcciones de seguridad que los protegerán contra ataques de malware.

Algunas personas siguen usando versiones antiguas de software y se niegan a actualizarlas. Normalmente, esto se debe a que la nueva versión rompe algo que funcionaba en su entorno. Cuando las empresas cobran dinero por las actualizaciones, algunos clientes se niegan a actualizar porque no quieren pagar dinero extra, pero pagar por las actualizaciones es poco común en el código abierto.

Los desarrolladores se adaptan a los usuarios de versiones antiguas, si es posible, corrigiendo errores en las versiones antiguas sin agregar características que rompan el software.

Naturalmente, los desarrolladores no pueden hacer esto para siempre porque consume tiempo y energía del nuevo trabajo. Llegará un momento en el que se negarán a reparar una versión antigua y les dirán a los usuarios que actualicen o que hagan sus propias correcciones.

La reparación de errores y fallas de seguridad se denomina *soporte* para el software. Esto es diferente del soporte que ofrecen los servicios de asistencia técnica y otras personas que guían a los usuarios para que comprendan el software. En lo que respecta a la gestión de versiones, una *versión compatible* es aquella en la que los desarrolladores prometen corregir errores, mientras que una *versión no compatible* es aquella en la que no lo hacen.

Tenga en cuenta que la noción de "soporte" se aplica principalmente al software creado por empresas u grandes organizaciones. Los proyectos de código abierto más pequeños, que dependen en gran medida de voluntarios, a menudo prometen simplemente hacer lo mejor que puedan para corregir errores y publicar nuevas versiones. No ven la necesidad de arreglar las versiones antiguas. Debido a que el código es abierto, por ende los usuarios que no quieran actualizar pueden pagarle a alguien para que arregle una versión anterior.

Cuando existe soporte, los desarrolladores publican un cronograma que indica durante cuánto tiempo brindarán soporte para cada versión. La fecha en la que finaliza el soporte se denomina *fin de vida útil* (EOL) del software. Los usuarios pueden mantener la versión anterior hasta el EOL y esperar que se solucionen los errores, pero después del EOL tienen que actualizar o arriesgarse.

Los proyectos importantes, como la distribución Debian de GNU/Linux, ofrecen versiones de soporte a largo plazo (LTS). Eso simplemente significa que los desarrolladores seguirán corrigiendo errores durante un cierto número de años. Los clientes que valoran mucho la estabilidad pueden tener miedo de instalar versiones de software que tengan muchos cambios, en caso de que los cambios rompan los procesos en los que confían los clientes. A estos clientes, en particular a las grandes instituciones, les gusta la seguridad de las versiones LTS.

Versiones de software: principales (Major), menores y parches

Hemos visto que el control de versiones es complicado, algunas versiones corrigen errores, mientras que otras agregan funciones y la estabilidad de las versiones puede variar. Los desarrolladores intentan indicar mediante etiquetas qué tan grande es el cambio en el software en cada versión. Las convenciones adoptadas por casi todos los proyectos para etiquetar versiones se denominan *versionamiento semántico*.

Tomemos como ejemplo la historia del kernel de Linux. Linus Torvalds etiquetó la primera versión estable como 1.0. A medida que mejoró el kernel, etiquetó las versiones posteriores 1.1, 1.2, etc. El 1 inicial era la *versión principal* y los números después del punto denotaban la *versión menor*. Se podría suponer que la versión 1.2 tenía más funciones y ofrecía más que la versión 1.1.

Pero también hubo innumerables versiones pequeñas, a veces sólo para corregir algunos errores. Para demostrar que los cambios tuvieron un impacto menor en el uso del kernel, Torvalds incluyó un tercer número llamado *patch*. Por lo tanto, 1.0 se actualizó a 1.0.1, luego a 1.0.2, y así sucesivamente.

Los números de parche comienzan desde 0 cuando se lanza una nueva versión secundaria, y los números de versión secundaria comienzan desde 0 cuando se lanza una nueva versión principal.

Los desarrolladores agrupan las versiones usando una “x” para indicar que están hablando de múltiples versiones, como 1.x para todas las versiones bajo la versión principal 1.

Pero, ¿cuál es la diferencia entre un cambio que conduce a una nueva versión menor y un cambio que merece desencadenar una nueva versión principal? Por lo general, pasan años antes de que se lance otra versión y ésta debe reflejar una actualización muy significativa.

Generalmente, los desarrolladores intentan mantener la compatibilidad con versiones anteriores a medida que cambian las versiones. Si no se puede preservar la compatibilidad con versiones anteriores, los desarrolladores deben incrementar la versión principal.

A veces ves un número de versión menor que cero, normalmente 0,9. El cero inicial indica una versión anterior del software que es inestable y no está lista para su uso en producción. No hay garantía de compatibilidad con versiones anteriores hasta que los desarrolladores publiquen versiones que comiencen con 1.

Las versiones alfa generalmente se indican agregando “a” o “alpha” después del número de versión, como 3.6alpha. De manera similar, las versiones beta agregan “b” o “beta” después del número de versión.

El ciclo de vida del producto de software

No creas que la vida de un desarrollador es fácil. Todo el mundo quiere algo de ellos. Quiero que este error en el diseño de la pantalla se solucione lo antes posible, mientras que alguien más dice que el error en el nombre de los archivos tiene prioridad. Los usuarios claman por nuevas características, y cuando diferentes desarrolladores trabajan en diferentes características en paralelo, descubren que los cambios de uno pueden pisotear los realizados por otro.

La gestión de proyectos y el subconjunto de estas tareas denominado gestión de versiones abordan estos problemas. Por lo general, un miembro senior del proyecto asume la responsabilidad de este trabajo de gestión.

Al igual que las personas, las versiones de software pasan por un ciclo de vida. Cada uno comienza con una discusión sobre las nuevas funciones y otros cambios necesarios, pasa por fases

de desarrollo y prueba, y se implementa de manera planificada.

Planificación y hojas de ruta (Roadmaps)

Los desarrolladores intentan establecer, con mucha antelación, qué cambios quieren hacer en un producto. En las empresas comerciales, hablan con gente de marketing, quienes filtran y resumen lo que les dicen sus clientes. Los proyectos de código abierto dependen más de las ideas enviadas por los desarrolladores y usuarios, que se capturan en una base de datos llamada *rastreador de problemas*. Si necesita corregir un error o desea una nueva función, complete un *problema*. Luego, los desarrolladores priorizan los cambios y deciden quién se encargará de cada uno.

Entonces, ¿cómo se eligen y priorizan los cambios? Puede ser un proceso complicado. Pero los buenos directores de proyectos en proyectos de código abierto fomentan una amplia aportación y al mismo tiempo garantizan que se tomen decisiones. Algunos proyectos incluso celebran conferencias en las que los participantes debaten las prioridades.

Una hoja de ruta publicada establece los planes resultantes para mejoras y cambios. Puede extender muchos lanzamientos y muchos años en el futuro. Cada paso se denomina *hito* y puede o no estar asociado con una fecha objetivo.

Programación de lanzamiento

Hay dos formas básicas de programar lanzamientos: según el tiempo y las funciones. Un proyecto puede prometer un lanzamiento a intervalos regulares (cada seis meses, por ejemplo) e incluir todo lo que esté terminado en ese momento. Alternativamente, el proyecto puede prometer incluir ciertas funciones en una versión y dejar que los desarrolladores se tomen el tiempo necesario para finalizar esas funciones.

A medida que se acerca el momento del lanzamiento, el administrador de lanzamientos determina las fechas para las versiones alfa, beta y estable. Los miembros del equipo revisan periódicamente los informes de errores e intentan planificar su trabajo para cumplir con estos hitos.

Para finalizar un lanzamiento, un proyecto debe dejar de aceptar ideas para nuevas funciones y centrarse en hacer que las funciones existentes funcionen correctamente. Este momento se llama *congelación de funciones*.

Documentación para las versiones del producto

Las hojas de ruta, como se mencionó anteriormente, explican las características que los desarrolladores planean incluir en cada versión. El lanzamiento va acompañado de una lista de

cambios denominada *changelog*. Generalmente, el registro de cambios enumera nuevas funciones, cambios, operaciones que se eliminaron, funciones que los desarrolladores pretenden eliminar en el futuro (conocidas como *obsoletas*) y correcciones de errores.

Por lo tanto, los registros de cambios pueden volverse bastante largos y detallados. Los usuarios deben prestar especial atención a las funciones que se eliminaron o que están en desuso, porque es posible que los usuarios tengan que cambiar sus programas o la forma de utilizar el producto. Obviamente, los desarrolladores intentan eliminar sólo aquellas funciones que ya nadie necesita.

Cada versión debe cambiar la documentación del producto para que coincida con los cambios en el producto. Esta tarea puede llevar mucho tiempo y es fácil para los desarrolladores pasar por alto un cambio o retrasarse en la producción de documentación.

Ejercicios guiados

1. ¿Qué distingue una versión estable de una inestable?

2. ¿Esperaría un cambio de funciones entre una versión denominada 2.6.14 y una versión denominada 2.6.15?

3. ¿Esperaría un cambio de funciones entre una versión denominada 2.6.0beta y una versión denominada 2.6.0?

4. ¿Por qué esperaría que la versión 1.0 no fuera compatible con la versión 0.9?

5. Si descubre una falla de seguridad en una versión después de una congelación de funciones pero antes de su lanzamiento, ¿puede solucionarla?

== Ejercicios exploratorios

6. Supongamos que desea seguir usando una versión de software de código abierto después del final de su vida útil, porque tiene una función que necesita. ¿Qué puedes hacer para mantenerlo utilizable?

7. ¿Cuáles son algunos de los criterios que permiten elegir una corrección de error o una solicitud de función sobre otras?

Resumen

Esta lección describió las principales características que distinguen las versiones: estabilidad, compatibilidad con versiones anteriores y soporte. Discutimos el significado de los nombres y números de las versiones, y los aspectos clave de la gestión de versiones, incluida la documentación.

Respuestas a ejercicios guiados

1. ¿Qué distingue una versión estable de una inestable?

   Las versiones se consideran estables de dos maneras: funcionan bien sin fallar ni producir resultados incorrectos, y se espera que la interfaz presentada a los usuarios o programadores sea compatible con versiones anteriores.

2. ¿Esperaría un cambio de funciones entre una versión denominada 2.6.14 y una versión denominada 2.6.15?

   No. El tercer número en el control de versiones semántico indica un parche que se creó para corregir un error o realizar alguna otra tarea menor, como reformatear. Un cambio de función debería dar lugar a una versión menor o mayor.

3. ¿Esperaría un cambio de funciones entre una versión denominada 2.6.0beta y una versión denominada 2.6.0?

   No. La versión beta es una versión de prueba que contiene todas las funciones que se incluirán en la versión final.

4. ¿Por qué esperaría que la versión 1.0 no fuera compatible con la versión 0.9?

   El número 0.9 advierte explícitamente a los usuarios potenciales que el software aún se está diseñando y que muy probablemente podría tener una interfaz diferente cuando se estabilice como versión 1.0.

5. Si descubre una falla de seguridad en una versión después de una congelación de funciones pero antes de su lanzamiento, ¿puede solucionarla?

   Sin duda. El período entre la congelación de una función y el lanzamiento es un momento para descubrir y corregir fallas, incluidas las de seguridad.

Respuestas a ejercicios exploratorios

1. Suponga que desea seguir usando una versión de software de código abierto después del final de su vida útil, porque tiene una función que necesita. ¿Qué puedes hacer para mantenerlo utilizable?

   Después del EOL, los desarrolladores del proyecto no tienen ningún compromiso de corregir errores, incluidos los fallos de seguridad. Por lo tanto, debe seguir atentamente el rastreador de errores y las listas de correo del proyecto para descubrir qué errores aparecen. Como el código está disponible, puedes y debes corregir los errores que se encuentran en tu versión. En teoría, incluso podrías incorporar nuevas funciones a tu versión. Eso efectivamente haría que su versión sea una bifurcación del original.

2. ¿Cuáles son algunos de los criterios que permiten elegir una corrección de error o una solicitud de función sobre otras?

   Para las correcciones de errores, los criterios incluyen la gravedad (que se asigna al error después de que ingresa al rastreador de errores) y la cantidad de usuarios afectados por él. Para una función, los criterios incluyen la cantidad de usuarios que desean la función, la dificultad de codificarla y su impacto potencial en otras partes del programa.

055.3 Community Management

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 055.3

Peso

2

Áreas de conocimiento clave

- Comprender los roles en proyectos de código abierto
- Comprender las tareas comunes en proyectos de código abierto
- Comprender los distintos tipos de contribuciones a proyectos de código abierto
- Comprender los distintos tipos de contribuyentes a proyectos de código abierto
- Comprender el papel de las organizaciones en el mantenimiento de proyectos de código abierto
- Comprender la transferencia de derechos de individuos a una organización que mantiene un proyecto
- Comprender las reglas y políticas en proyectos de código abierto
- * Comprender la atribución y la transparencia de las contribuciones Comprender aspectos de diversidad, equidad, inclusión y no discriminación

Lista parcial de archivos, términos y utilidades

- Desarrollo de software
- Documentación
- Diseños y obras de arte
- Soporte al usuario

- Desarrolladores
- Gerentes de lanzamiento
- Usuarios
- Líderes de proyectos y dictadores benévolos
- Individuos y corporaciones
- Entusiastas y profesionales
- Miembros del equipo central y colaboradores ocasionales
- Contribuciones de código y documentación
- Informe de errores
- Bifurcarción
- Fundaciones y patrocinadores
- Convenios de aportación
- Certificados de origen de desarrollador (DCO)
- Directrices de codificación
- Códigos de conducta

Linux Professional Institute

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	055 Gestión de Proyectos
Objectivo:	055.3 Gestión comunitaria
Lección:	1 of 1

Introducción

Una razón clave para crear un proyecto de código abierto es obtener contribuciones de muchas personas diferentes. El código abierto facilita responder a las diferentes necesidades e intereses de los usuarios del proyecto y beneficiarse de sus variadas habilidades. Por lo tanto, una comunidad diversa es fundamental para el éxito del proyecto. La comunidad es donde las fuerzas creativas se unen para que tu proyecto tenga éxito.

Esta lección explica los elementos básicos del software libre y las comunidades de código abierto, y cómo ellas trabajan juntas. Por supuesto, como las comunidades están formadas por diferentes personas y los proyectos tienen diferentes objetivos y requisitos, cada comunidad es única.

Roles en proyectos de código abierto

El principal resultado de la mayoría de los proyectos de código abierto es el software, por lo que, como mínimo, dichos proyectos requieren programadores, diseñadores o arquitectos de software, evaluadores, administradores de versiones y otros expertos en desarrollo de código. La documentación también suele formar parte de estos proyectos, por lo que la comunidad también necesita autores, editores y revisores.

Es posible que otros proyectos no produzcan código, sino otros “entregables”, como un libro, un proyecto de arte o música, o un informe de políticas. Wikipedia es un ejemplo bien conocido de colaboración en un proyecto de código abierto que se centra en documentos y medios de texto. Estos proyectos necesitan expertos en el diseño, producción y entrega del entregable.

Las comunidades también se benefician de muchas otras habilidades generales, como marketing y evangelización, administración, asesoramiento legal, talento artístico para producir logotipos o diagramas en la documentación y habilidades para mantener el sitio web del proyecto y otras herramientas de comunicación. Los proyectos de código abierto pueden organizar reuniones físicas e incluso conferencias, en cuyo caso necesitan organizadores que den vida a esos eventos.

Los párrafos siguientes le dan una idea de los tipos de habilidades que busca una comunidad. Estos roles generales se pueden subdividir en varias tareas enfocadas. Un ejemplo sencillo de tal tarea es que algunos escritores editen el trabajo de otros escritores.

Entre los programadores existe una variedad de experiencias. Un proyecto saludable siempre necesita algunos programadores *veteranos* (o programadores *senior*) que conozcan bien su código y sus estándares de codificación, junto con *novatos* que proporcionen nuevas ideas y ayuda sencilla, como corrección de errores. Es de esperar que algunos novatos se conviertan en la próxima generación de veteranos.

La calidad del código requiere un control cuidadoso sobre lo que ingresa al repositorio central compartido con los usuarios. Por lo tanto, algunos veteranos obtienen el estatus de *committer*. Son responsables de verificar la calidad, utilidad y conformidad de las contribuciones con los estándares de codificación. Tienen los privilegios de aceptar el código propuesto en el repositorio confiable. Otros contribuyentes envían código a los confirmadores (committers) para su revisión.

Muchos confirmadores y otros veteranos también asesoran a nuevos contribuyentes para enseñarles los estándares y prácticas necesarios para que su código sea aceptado.

Algunos contribuyentes no aprecian el trabajo que ofrece un colaborador. Si la contribución no cumple con los estándares de calidad o codificación, el autor podría simplemente rechazarla. Pero el rechazo desanima al posible contribuyente y elimina una valiosa oportunidad educativa. Recuerde que los buenos confirmadores también son mentores. Por lo tanto, le dan al colaborador sugerencias sobre cómo llevar el código a los estándares y trabajar con el colaborador a lo largo del tiempo. Si la contribución es realmente inutilizable, al menos se debe agradecer al contribuyente y alentarlo a trabajar en otra cosa para el proyecto. Es importante ayudar a las personas a descubrir y perfeccionar sus habilidades.

Cuando una empresa inicia un proyecto de código abierto, a veces nombra responsables de su propio personal para garantizar que pueda controlar la dirección y la calidad del proyecto. Pero a menudo, las empresas permiten que personas que no son empleados y que demuestran habilidad

y dedicación, se comprometan.

Los *líderes de proyectos* toman decisiones críticas, como qué nuevas funciones admitir. Estos líderes de proyecto a menudo también tienen que tomar decisiones no relacionadas con la codificación, como cómo promover el proyecto, resolver desacuerdos importantes y obtener financiación. Los líderes de proyecto y los encargados de confirmar trabajan en estrecha colaboración con los *gerentes de lanzamiento* para decidir cuándo una base de código es estable y está lista para compartir con los usuarios.

Algunos proyectos tienen un único líder, a menudo denominado “dictador benévolente”. Suele ser una persona que inició el proyecto (Linus Torvalds es un ejemplo muy conocido en el caso de Linux) y que tiene una gran autoridad e incluso carisma. Sin embargo, la mayoría de los proyectos prefieren establecer un pequeño comité de líderes de proyecto en lugar de un único dictador benévolente. Incluso el proyecto Linux ha pasado a adoptar un enfoque de comité.

Los proyectos también pueden solicitar a contribuyentes particulares que brinden soporte a los usuarios en los foros del proyecto. Pero los proyectos saludables deberían tener muchos usuarios informados que se apoyen entre sí.

Terminaremos esta sección con un rol muy importante: el *community manager*. Es alguien responsable de mantener una comunicación abierta y constructiva y de asegurarse de que la comunidad avance hacia su objetivo. El administrador de la comunidad sabe cómo incorporar y motivar a los contribuyentes, resolver discusiones, proteger a los miembros de la comunidad contra el abuso y realizar otras tareas para mantener la salud de la comunidad.

Tareas comunes en proyectos de código abierto

En muchos sentidos, los proyectos de código abierto implican las mismas tareas que los proyectos tradicionales llevados a cabo dentro de una única organización. Por ejemplo, todo el código requiere pruebas. Pero en algunos aspectos, los proyectos de código abierto difieren de los propietarios, donde sólo un grupo limitado de personas pueden cambiar el código y donde dicha fuente a menudo está oculto al público.

Por ejemplo, las corporaciones suelen asignar personal capacitado en control de calidad (QA) para probar el código. Pero muchos proyectos de código abierto simplemente piden a sus usuarios que prueben cada versión. (Los proyectos propietarios también involucran a sus usuarios en pruebas, además del control de calidad).

Otro ejemplo de diferencias: un proyecto propietario generalmente asigna pagos a los desarrolladores por tareas específicas y les dice cuánto tiempo cada semana dedicar a esas tareas. Algunos proyectos de código abierto se benefician de contribuyentes pagados por sus

empleadores, o en ocasiones incluso empleados por el proyecto mismo, y a quienes se les asignan tareas de la misma manera que los desarrolladores propietarios. Pero en la mayoría de los proyectos saludables, muchas contribuciones provienen de voluntarios que simplemente realizan las tareas cuando el tiempo lo permite. Debido a que el proyecto no depende de que todos los voluntarios completen sus proyectos a tiempo, un lanzamiento de código abierto podría retrasarse hasta que los desarrolladores sientan que las funciones están listas, o podría publicarse en momentos fijos con las funciones que estén listas.

Las comunicaciones son fundamentales tanto para proyectos propietarios como para proyectos de código abierto, pero a menudo se llevan a cabo de manera diferente. Los proyectos propietarios a menudo reúnen a un equipo en una oficina y celebran reuniones periódicas programadas bajo una metodología de desarrollo como SCRUM. Debido a que los proyectos de código abierto están distribuidos geográficamente, este tipo de metodologías son raras. En cambio, la comunicación se produce en línea y de forma asincrónica.

En resumen, los proyectos de software libre y de código abierto a menudo logran los mismos objetivos que los proyectos propietarios, pero de diferentes maneras.

La notificación de errores es una parte fundamental del desarrollo de software que da lugar a su propio conjunto de funciones. Alguien tiene que monitorear la base de datos de errores y asignar prioridades. Si un error es crítico (y particularmente si puede debilitar la seguridad del software), se debe asignar a un mantenedor competente para que lo solucione.

Los líderes de proyecto y los administradores comunitarios deben revisar la participación en el proyecto y ver dónde hay brechas. ¿Hay muy pocas personas dispuestas a probar el código? ¿Falta documentación? ¿Se retiran demasiados veteranos o miembros de alto nivel del proyecto sin ser reemplazados? Los líderes de proyecto y los administradores de la comunidad pueden concentrarse en reclutar personas para desempeñar los roles necesarios.

Los líderes de proyecto y los administradores de la comunidad en proyectos grandes a menudo también recopilan métricas para aprender cosas importantes, como si los errores importantes se están solucionando de manera oportuna.

Tipos de contribuciones de código abierto

Como se explicó en una sección anterior, las personas que colocan código, documentación u otros elementos en el repositorio central se denominan *contribuidores*. Los miembros del proyecto que aprueban las contribuciones y las aceptan en el repositorio se denominan *contribuyentes*. Algunos miembros asumen otras funciones comunes para el desarrollo de software.

Aunque el término *contribuidor* generalmente está reservado para alguien que desarrolla código

u otra parte de la oferta oficial del proyecto, cualquiera que participe en el éxito del proyecto está contribuyendo de alguna manera y se le debe agradecer por hacerlo. Estas contribuciones más informales incluyen informar un error, responder una pregunta en un foro, donar o recaudar fondos y promover el proyecto entre personas externas.

Los proyectos de código abierto, como los propietarios, preguntan a los usuarios qué quieren en términos de características, mejoras de rendimiento u otros cambios en el proyecto. Estos usuarios están haciendo contribuciones importantes al expresar sus opiniones.

Tipos de contribuyentes de código abierto

Muchas comunidades obtienen contribuciones no sólo de individuos, sino también de corporaciones que pagan a su personal para que contribuyan a un proyecto que la corporación considera importante para su negocio.

A veces, tener contribuyentes remunerados junto a los voluntarios puede crear tensiones. Los voluntarios pueden sentir que su trabajo está siendo explotado, o pueden temer que los contribuyentes remunerados estén tratando de desviar la dirección del proyecto para favorecer los intereses de sus empleadores. Un community manager y los líderes del proyecto deben garantizar que todas las contribuciones aceptadas proporcionen beneficios a todo el proyecto y a sus usuarios. Los voluntarios también deben recibir motivación para contribuir por motivos personales, ya sea por lealtad a la comunidad o para satisfacer sus necesidades.

Algunas personas contribuyen a un proyecto con regularidad y asumen responsabilidades a largo plazo; estos se denominan *miembros del equipo central*. Las comunidades saludables también tienen contribuyentes ocasionales que informan errores o publican consejos en foros, pero no se espera que asuman la responsabilidad del proyecto.

De manera similar, algunos contribuyentes serán profesionales en su campo (ya sea que la empresa les pague o no por trabajar en el proyecto), mientras que otros son aficionados, a menudo llamados *entusiastas*. Pero no es necesario ser un profesional para asumir un papel importante. Incluso podría convertirse en miembro central del equipo o líder de proyecto.

El papel de las organizaciones en proyectos de código abierto

Aunque muchos proyectos de código abierto son creados por personas entusiastas o pequeños grupos de voluntarios, normalmente buscan apoyo organizacional cuando los proyectos crecen. El apoyo organizacional puede tomar muchas formas. En general, las organizaciones hacen estas contribuciones porque un proyecto de código abierto puede satisfacer sus necesidades comerciales de manera más económica y confiable que reinventar las mismas características en

código propietario. Muchas organizaciones valoran las garantías que ofrece una licencia libre o de código abierto.

Algunos idealistas desconfían de las corporaciones u otras grandes organizaciones y preferirían mantener los proyectos de código abierto completamente libres de su influencia. Con el paso de los años, esta actitud bastante simplista se ha vuelto menos común. La mayoría de los defensores del código abierto creen que las corporaciones y otras organizaciones pueden proporcionar bases cruciales para los proyectos de código abierto.

Muchos proyectos de código abierto comienzan dentro de una organización e incluso pueden basarse en código propietario que la empresa decide abrir. Para ganar dinero, la empresa puede decidir mantener un control firme sobre el proyecto y dividirlo en partes abiertas y propietarias: esto se denomina modelo de núcleo abierto. Muchos fundadores de proyectos comienzan como código abierto pero forman una empresa a su alrededor, que puede o no utilizar el modelo de núcleo abierto.

Otros proyectos intentan garantizar que ninguna empresa controle su dirección. Mantener la independencia generalmente requiere reclutar a varias organizaciones que la apoyen, de modo que ninguna sea lo suficientemente fuerte como para tomar una decisión dictatorial.

¿Cómo apoyan las organizaciones el código abierto? Como se mencionó anteriormente, muchos asignan personal remunerado a los proyectos. Además, podrían contratar personas altamente productivas que hayan contribuido al proyecto como voluntarios y hayan desarrollado experiencia en el proyecto. Este camino hacia el empleo es una forma valiosa para que los estudiantes y otros contribuyentes voluntarios avancen en sus carreras.

Las empresas que deseen extensiones que no interesen a otros usuarios no deberían presionar al proyecto de código abierto para que agregue características adicionales, sino que deberían pagar a su personal para que escriba las características sin contribuir con ellas al proyecto.

Un ejemplo interesante de la posible tensión creada por las necesidades corporativas lo muestra el famoso sistema operativo Android, basado en Linux y desarrollado por Google para impulsar sus dispositivos móviles. Algunos cambios en Linux realizados por Google se aportan a la comunidad de Linux, mientras que otros cambios aparecen sólo en Android. Los desarrolladores de Linux a veces rechazan los cambios que les envía Google, del mismo modo que todos los proyectos eligen qué contribuciones incorporar.

Hay muchas razones para que una empresa o un grupo de desarrolladores creen una versión separada de su código. Idealmente, pueden satisfacer sus necesidades agregando una biblioteca opcional o una secuencia de código que pueda excluirse durante la compilación. Pero a veces un grupo siente una necesidad importante que es incompatible con la dirección elegida por los líderes del proyecto. Cuando se crea una versión separada, se llama *fork*.

Las bifurcaciones solían considerarse fracasos de colaboración, pero hoy en día son mucho más aceptadas. Por lo general, las personas que crean una bifurcación, configuran un nuevo repositorio y comienzan un nuevo proyecto. Algunas personas trabajan tanto en el proyecto original como en el fork.

(Tenga en cuenta que GitHub usa la palabra “fork” para un fenómeno muy diferente: hacer un clon o una copia del código del proyecto para trabajar por separado).

Además del código, muchas empresas contribuyen económicamente. Pueden financiar esfuerzos como marketing y conferencias. Pueden unirse a la junta y ofrecer asesoramiento experto sobre la dirección y estrategia del proyecto.

Un ejemplo de la importancia de este “soporte suave” (soft support) proviene del histórico servidor web Apache. El proyecto dio un gran paso adelante al principio de su existencia cuando IBM mostró interés. Los abogados de IBM mostraron al proyecto cómo crear salvaguardias legales, y una reunión pagada por IBM ayudó a los líderes de Apache a crear una organización sólida.

Para mantener la independencia, muchos proyectos de código abierto forman una fundación sin fines de lucro o se unen a una fundación existente. Ejemplos famosos de fundaciones que guían proyectos de código abierto incluyen la Fundación Linux, la Fundación Apache y la Fundación Eclipse.

El apoyo de una fundación es valioso porque puede proporcionar la logística con la que la mayoría de los desarrolladores de software no quieren lidiar: apoyo legal como marcas registradas, indemnización y licencias, ayuda para recaudar dinero, infraestructura como bases de datos de errores y sitios web, etc.

Cesión de Derechos

Al igual que ocurre con los libros, la música y otros esfuerzos creativos, el software implica una relación complicada entre los desarrolladores, las organizaciones que distribuyen sus contribuciones y el público en general. Esencialmente, los contribuyentes deben tomar medidas para formalizar el derecho de un proyecto de código abierto a utilizar y distribuir su código.

Por lo tanto, muchas organizaciones de código abierto solicitan a sus contribuyentes que firmen un acuerdo de licencia de colaborador (CLA). A veces, el colaborador simplemente proporciona el código del proyecto. El proyecto es propietario del código y de todos los derechos sobre él, del mismo modo que una empresa propietaria es propietaria del código cuyo desarrollo pagó a su personal.

Otros CLA dejan algunos derechos en manos de los contribuyentes individuales. A los contribuyentes les puede gustar esta flexibilidad porque pueden contribuir con el mismo código a

otro proyecto o construir su propio negocio en torno a él.

Para armonizar las diferentes contribuciones de diferentes personas, la licencia asignada al código es crucial. El kernel de Linux es uno de los proyectos que deja la propiedad en manos de los contribuyentes, de modo que ahora miles de personas poseen los derechos sobre el código de Linux. Pero todos los contribuyentes al código central lo colocaron bajo la Licencia Pública General GNU (versión 2). Por lo tanto, Linux es libre para que todos lo usen, modifiquen y redistribuyan.

Usar una licencia única para todo el código de un proyecto es la forma más sencilla de garantizar que no haya obstáculos que impidan la distribución y el uso del código. Pero a veces un proyecto permite que se contribuyan diferentes partes de código bajo diferentes licencias, generalmente porque el proyecto quiere aprovechar algún código preexistente que ya está publicado bajo una licencia diferente. Los expertos en licencias deben asegurarse de que las estas sean compatibles.

Reglas y Políticas

Muchas comunidades en línea tienen fama de expresarse sin restricciones (una idea arrogante de que “todo vale”) que conduce a abusos verbales y disputas. Hoy en día, la mayoría de las comunidades de código abierto están luchando contra esa tendencia, que se apodera de la gente con demasiada facilidad. Por el contrario, las comunidades modernas quieren que las personas sean constructivas, cívicas, respetuosas e inclusivas con todos: géneros, grupos étnicos, tipos de personalidad, etc.

Las expectativas de los miembros del grupo suelen especificarse explícitamente en un código de conducta que explica cómo interactuar con otras personas en el proyecto, tanto en línea como en persona. Para que sea eficaz, este código de conducta debe ser aplicado por el administrador de la comunidad y los líderes del proyecto.

En ocasiones, una persona que se burla o denigra flagrantemente a un compañero de la comunidad es expulsada de forma permanente o temporal. En otras ocasiones, el administrador de la comunidad o un colega simplemente anuncia públicamente que el comportamiento viola el código de conducta y podría hablar con el infractor fuera del foro. A menudo, el agresor previamente sintió frustración, falta de atención o agotamiento, y los miembros de la comunidad pueden ayudarlo a encontrar mejores maneras de expresarse.

Además del comportamiento social, las comunidades establecen estándares de calidad. Las más formales consisten en *pautas de codificación*, que intentan garantizar que todo el código sea similar. Las pautas pueden recordar a los desarrolladores buenas prácticas como el uso de corchetes alrededor de bloques de código, o pueden especificar detalles como cómo nombrar las variables y qué tipo de sangría usar.

Las comunidades de código abierto también tienen reglas sobre la publicación de código u otros entregables. Algunos proyectos establecen tiempos fijos para las liberaciones; por ejemplo, Ubuntu promete una nueva versión de soporte a largo plazo (LTS) cada dos años junto con lanzamientos provisionales con mayor frecuencia. Otros proyectos mantienen una lista de nuevas funciones y correcciones de errores que desean en la próxima versión y aprueban la versión cuando todo está marcado en la lista. Pero a diferencia de la mayoría de las empresas, las comunidades de código abierto no suelen establecer plazos para los participantes, porque no se puede pedir demasiado a los voluntarios.

Atribución y Transparencia

Además del acuerdo de licencia de contribuyente mencionado anteriormente, algunos proyectos solicitan a los contribuyentes que firmen un *certificado de origen de desarrollador* (DCO). En el DCO, el contribuyente promete que tiene el derecho legal de donar el código.

¿Por qué es importante el DCO? Imagine este escenario: si un programador toma código de un producto propietario producido por su empleador y lo aporta a un proyecto de código abierto, el programador viola la licencia del empleador y pone el proyecto de código abierto en riesgo legal.

Se supone que el DCO debe garantizar que el contribuyente realmente haya escrito el código o lo haya obtenido legalmente. El proyecto de código abierto depende de la honestidad del colaborador que completa el certificado.

Diversidad, Equidad, Inclusividad y No Discriminación

Los científicos sociales afirman que los proyectos y las empresas se benefician al tener muchas personas con diferentes géneros, razas, orígenes económicos, nacionalidades y habilidades. Todas las organizaciones tienen una tendencia humana natural a vincularse con otras personas como sus miembros actuales, por lo que una comunidad que valora la diversidad y la equidad tiene que capacitar conscientemente a sus miembros para que sean más abiertos a las personas que no son como ellos. El movimiento para llevar a cabo este ideal se llama diversidad, equidad e inclusión (DEI).

El código de conducta es el punto de partida de DEI. Debería dar la bienvenida explícitamente a personas de diferentes géneros, razas, etc. Cada violación del código de conducta debe abordarse con prontitud y con una desaprobación inequívoca. Esto se debe a que algunas minorías han sufrido exclusión y comentarios negativos toda su vida, y una sola mala interacción en su foro podría hacerles decidir que ya no tienen motivos para participar. Una respuesta rápida a las agresiones puede asegurarles que la comunidad los respalda.

Pero DEI va mucho más allá. La comunidad debería acercarse a los grupos que necesitan más

representación. Por ejemplo, si está desarrollando una aplicación de búsqueda de empleo, debe asegurarse de que muestre trabajos para personas de bajos ingresos, así como para personas de clase alta y media. También debes promocionar la aplicación entre las comunidades de bajos ingresos y reclutar miembros de esas comunidades para que la prueben.

Su comunidad podría beneficiarse al encontrar organizaciones que capaciten a personas de comunidades marginadas y de las cuales pueda reclutar nuevos miembros para su equipo. Muchas de estas organizaciones están establecidas en áreas locales y no son muy conocidas a nivel nacional o internacional.

Comprender las necesidades de las comunidades marginadas es clave. ¿Su sitio web es accesible para personas con discapacidad visual? ¿Traduce su documentación a idiomas con los que la comunidad de destino está familiarizada? Quizás deberías crear foros específicos en otros idiomas.

La mayor parte de la comunicación en las comunidades de código abierto es asincrónica y en línea. Pero si tiene reuniones o sesiones de chat sincrónicas, piense dónde están geográficamente las personas y encuentre formas de incluirlos a todos. Por ejemplo, cuando personas de muchos países y regiones se comunican en inglés, trate de mantener el vocabulario y la gramática simples.

Finalmente, si tienes miembros de alguna minoría en tu equipo, asegúrate de escucharlos. Un síntoma bien conocido de exclusión es descartar lo que dicen las minorías o simplemente subestimar su importancia.

Por otro lado, no cargue a los miembros de las minorías con explicar las necesidades de sus comunidades: todos deberían tener la tarea de esa investigación. Los miembros de la minoría podrían hacer una valiosa labor de divulgación para usted, pero no los presione para que lo hagan. Cada persona debe tener las mismas oportunidades de participar como quiera, sin tener que ser una minoría simbólica.

Ejercicios guiados

1. ¿Por qué no todos los contribuyentes dan su código al proyecto y renuncian a todos los derechos sobre el código?

2. Si alguien quiere ayudar en el proyecto pero no sabe programar, ¿de qué maneras puede ayudar?

3. ¿Por qué muchos proyectos de código abierto se unen a una fundación?

Ejercicios exploratorios

1. Eres un comprometido con tu proyecto. Alguien envía un código que se tomó de otro proyecto (pero el colaborador tiene los derechos sobre él). El código tiene un formato completamente diferente al resto del código. ¿A qué te dedicas?

2. Creó un producto de software propietario y desea contribuir con partes del mismo a un proyecto de código abierto. ¿En qué circunstancias puede seguir ofreciendo su producto patentado?

3. Dos personas de su lista de correo comienzan a discutir sobre una característica de su código. La discusión se vuelve gradualmente más acalorada hasta que una persona llama idiota a la otra. ¿Cómo puedes manejar la situación?

Resumen

En esta lección, aprendió lo que es ser parte de una comunidad y cómo las comunidades siguen siendo productivas. Aprendió diferentes tipos de contribuciones, contribuyentes y roles. Aprendiste cómo las comunidades controlan el derecho a utilizar las contribuciones. Aprendió sobre las reglas de una comunidad y cómo protegen a todos, incluidas las personas de diferentes razas y géneros, del abuso verbal.

Respuestas a ejercicios guiados

1. ¿Por qué no todos los contribuyentes dan su código al proyecto y renuncian a todos los derechos sobre el código?

   Es posible que el colaborador desee contribuir con el mismo código a un proyecto diferente o lanzar un producto basado en el código y, por lo tanto, desee conservar algunos derechos.

2. Si alguien quiere ayudar en el proyecto pero no sabe programar, ¿de qué maneras puede ayudar?

   Hay muchas funciones para los contribuyentes además de la codificación. Algunas de estas funciones incluyen documentación, pruebas e informes de errores, gestión de la comunidad, participación en formularios, promoción del proyecto y creación de obras de arte.

3. ¿Por qué muchos proyectos de código abierto se unen a una fundación?

   Una fundación maneja muchas tareas legales, financieras y de otro tipo que la comunidad del proyecto podría tener dificultades para realizar.

Respuestas a ejercicios exploratorios

1. Eres un comprometido con tu proyecto. Alguien envía un código que se tomó de otro proyecto (pero el colaborador tiene los derechos sobre él). El código tiene un formato completamente diferente al resto del código. ¿A qué te dedicas?

   Los estándares de codificación de tu proyecto deben explicar claramente cómo formatear el código. Agradezca al colaborador, indíquele los estándares y pídale que vuelva a formatear el código. A veces, existen herramientas automatizadas para reformatear el código como desee. Si el colaborador no tiene tiempo para reformatear, busque un miembro junior de su equipo que pueda hacer ese trabajo.

2. Creó un producto de software propietario y desea contribuir con partes del mismo a un proyecto de código abierto. ¿En qué circunstancias puede seguir ofreciendo su producto patentado?

   La respuesta depende del acuerdo de licencia del colaborador. Si el CLA requiere que usted entregue el código al proyecto, otorgándole todos los derechos, es posible que no pueda continuar ofreciendo su producto propietario. Sin embargo, si le permiten conservar sus derechos sobre el código, puede liberarlo bajo cualquier licencia que desee en su producto propietario.

3. Dos personas de su lista de correo comienzan a discutir sobre una característica de su código. La discusión se vuelve gradualmente más acalorada hasta que una persona llama idiota a la otra. ¿Cómo puedes manejar la situación?

   Cualquiera que note la escalada y el comentario abusivo debe reaccionar lo más rápido que pueda. El community manager es el responsable último de reparar los daños. La persona que interviene debe publicar un comentario en toda la lista diciendo que el comportamiento viola el código de conducta del proyecto (que con suerte descarta dicho comportamiento). La persona que interviene también puede comunicarse con las personas de cada lado por separado para asegurarse de que estén satisfecho con la resolución del argumento y entiende cómo discutir los desacuerdos de manera constructiva.

Linux Professional Institute

Tema 056: Colaboración y comunicación

056.1 Herramientas de desarrollo

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 056.1

Peso

2

Áreas de conocimiento clave

- Comprender las herramientas comunes de desarrollo de software
- Comprender los entornos de implementación comunes
- Comprender los tipos comunes de pruebas de software
- Comprensión de los conceptos de integración continua y entrega continua (CI/CD)

Lista parcial de archivos, términos y utilidades

- Entornos de desarrollo integrados (IDE)
- Linters
- Compiladores
- Depuradores
- Ingeniería inversa
- Refactorización
- Examen de la unidad
- Pruebas de integración
- Test de aceptación
- Pruebas de rendimiento

- Prueba de humo
- Pruebas de regresión
- Sistemas de producción, puesta en escena y desarrollo.
- Sistemas de desarrollo local
- Sistemas de desarrollo remoto
- Canalizaciones de CI/CD

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	056 Colaboración y comunicación
Objectivo:	056.1 Herramientas de desarrollo
Lección:	1 de 1

Introducción

Existen miles de herramientas de desarrollo de software, tanto de código abierto como de código propietario. ¿Y por qué no? A los programadores les encanta desarrollar herramientas para ellos mismos y para sus colegas; es natural que inviertan mucho tiempo en intentar encontrar una herramienta que funcione mejor, que elimine algunas molestias de su flujo de trabajo o facilite la implementación.

Esta lección se centra en el proceso de desarrollo y explica cómo encajan en él los distintos tipos de herramientas de desarrollo. Se nombrarán algunas herramientas específicas, ya que cualquiera de ellas podría estar marcada como obsoleta o en desuso y ser reemplazada por una nueva cuando lea esta lección.

Objetivos del desarrollo

Las herramientas de programación combinan múltiples objetivos que a veces entran en conflicto entre sí. A continuación, se muestran ejemplos de algunos objetivos de desarrollo:

- Producir programas robustos y precisos.

- Producir programas que se ejecuten rápidamente.
- Producir programas que puedan escalarse.
- Producir programas que sean altamente adaptables a diferentes tipos de usuarios, diferentes dispositivos (como computadoras portátiles, teléfonos celulares y tabletas) y diferentes entornos.
- Acelerar el proceso de desarrollo.
- Acelerar la implementación de funciones recientemente desarrolladas o correcciones de errores.
- Reducir el trabajo tedioso, así como la cantidad de errores de programación administrativa que pasan a las etapas de prueba..
- Admitir códigos heredados y dispositivos que la organización tiene dificultades en reemplazar.
- Trabaje junto con otras herramientas conocidas.
- Permitir una fácil reversión en caso de errores o cambios en los planes.

Estos objetivos, y otros, conducen a los procesos descritos en esta lección y a las herramientas de desarrollo resultantes.

Procesos generales de desarrollo

Esta sección contrasta dos modelos generales de importancia histórica: el modelo en cascada (introducido en una lección anterior) y la integración continua/entrega continua (CI/CD).

Modelo de cascada

Aunque el modelo en cascada todavía se utiliza ampliamente, especialmente en las grandes organizaciones, ha caído en desuso entre muchos desarrolladores. Este modelo fue popular desde la década de 1950 hasta la de 1970. Algunos elementos del modelo todavía se pueden encontrar ampliamente en la actualidad, como las definiciones formales de los requisitos y los programas de prueba justo antes de su lanzamiento (*garantía de calidad*).

Incluso cuando se adoptó el término “cascada” para este modelo, ya había quedado ampliamente desacreditado. Entre los problemas que presentaba se encontraban los siguientes:

- Cambiar los requisitos fue difícil una vez que comenzó el desarrollo.
- Los requisitos y diseños se malinterpretaban a menudo porque el texto en lenguaje sencillo puede ser ambiguo. Este problema dio lugar a productos que no cumplían los requisitos previstos y tardaban meses en solucionarse.

- El proceso era lento. Se podía conseguir un nuevo lanzamiento solo una vez al año o incluso con menos frecuencia.
- Los errores causados por la interacción de diferentes módulos eran difíciles de detectar hasta una etapa avanzada del proceso, lo que eventualmente se convertía en más demoras.
- El proceso creó barreras entre las diferentes partes de la organización, lo que fue malo para la calidad del producto y para el espíritu del cuerpo organizacional.

Principios de integración continua/entrega continua (CI/CD)

El modelo en cascada fue desafiado en la década de 1970 por una serie de movimientos que llevaron al modelo más popular (si no universalmente utilizado) en la actualidad: CI/CD. Las etapas de la adopción de las prácticas de CI/CD incluyen el *Manifiesto Ágil*, publicado en 2001, SCRUM y DevOps. El nuevo modelo se basa en los principios de una comunicación estrecha entre los miembros del equipo, la participación del usuario final o cliente, la rápida implementación de correcciones de errores y nuevas características, pruebas rigurosas y continuas para preservar la calidad en un entorno de rápido movimiento, a veces incluso caótico.

La CI/CD automatiza tantos pasos como sea posible en las etapas que van desde la escritura del código hasta la implementación del programa completo para los usuarios finales. La CI/CD requiere definir cada paso formalmente y reemplazar una acción humana (como instalar software manualmente) con un procedimiento ejecutado por un programa. Por lo tanto, la CI/CD es parte de un movimiento conocido con la frase “todo como código” e “infraestructura como código”.

Además, una vez que un equipo ha incorporado sus procesos al código, estos procesos se pueden corregir, actualizar y registrar históricamente, al igual que el código. Todo lo que escriba el equipo, ya sean programas, procedimientos automatizados o documentación, debe almacenarse en un sistema de control de versiones, que se describe en una próxima lección.

Cuando se automatizan varios procedimientos, estos pueden ejecutarse en secuencia. Por ello, los equipos suelen hablar de un *canal de CI/CD*, lo que significa que un paso exitoso en el canal desencadena automáticamente uno o más procesos posteriores. Un canal debe supervisarse de forma automatizada para que cualquier fallo en el camino provoque que el canal se detenga y notifique a los miembros del equipo sobre el fallo.

Aunque en las siguientes secciones se analizan el CI y la CD por separado, tienden a combinarse y la línea divisoria entre ellas es borrosa.

Integración Continua (IC)

La *integración continua* se refiere a la rápida incorporación de pequeños cambios en el código. El repositorio central que se utiliza para crear el producto para los usuarios finales se conoce como *repositorio central* (o *repositorio principal*). Cada programador crea un espacio de trabajo personal (a menudo llamado *sandbox*) en su computadora y extrae del repositorio central los archivos que necesita para reparar o actualizar.

El programador podría utilizar una computadora personal portátil o de escritorio (lo que se conoce como *sistema de desarrollo local*), descargar las partes relevantes del repositorio central y luego cargar los cambios después de realizar pruebas locales. Como alternativa, el programador podría aprovechar la tendencia popular denominada “computación en la nube” y trabajar en un sistema compartido administrado por su organización, un *sistema de desarrollo remoto*.

El programador generalmente busca errores mediante un depurador, un programa independiente que ejecuta el trabajo del programador en un entorno controlado. Más adelante analizaremos los depuradores y otras herramientas para detectar errores.

Para detectar errores lo antes posible en el proceso de desarrollo, el programador también ejecuta pruebas en el código antes de subirlo al repositorio principal. Estas pruebas se denominan *pruebas unitarias* porque cada una se centra en un elemento minúsculo del código: por ejemplo, ¿una función incrementó un contador como se suponía que debía hacerlo?

La última etapa de la integración continua consiste en comprobar los cambios que el programador ha realizado en el repositorio central. En esta etapa, las herramientas ejecutan pruebas de integración para asegurarse de que el programador no haya dañado nada en el proyecto conjunto.

No importa lo lejos que haya llegado la automatización, algún miembro experto del equipo debe intervenir en el punto de integración para asegurarse de que el cambio sea el deseado por el equipo. Las pruebas automatizadas pueden determinar que nada se ha estropeado e incluso si el cambio crea el efecto deseado en el programa. Pero estas pruebas no pueden comprobar todos los principios que el equipo considera importantes.

Por lo tanto, antes de proceder a la implementación, un miembro del equipo debe verificar la seguridad, el cumplimiento de los estándares de codificación, la documentación adecuada y otros principios. (No es de sorprender que también existan herramientas para algunas de estas tareas; las analizaremos más adelante en esta lección).

Durante la integración continua se realizan muchas pruebas, que deben realizarse de forma rápida y fiable para soportar el ritmo del desarrollo moderno. Por lo tanto, las herramientas modernas para integrar código y ejecutar pruebas están automatizadas. Un programador debería

poder ejecutar un conjunto completo de pruebas unitarias con un solo comando o con el clic de un botón.

Generalmente, se ejecuta una prueba de integración parcial o total cada vez que el programador carga código en el repositorio principal. Cualquier error detectado por las pruebas se puede informar rápidamente al programador.

Entrega continua (CD)

Como se explicó en la sección anterior, la CI consiste en procedimientos automatizados que conducen a una nueva versión del programa en el directorio central. La *entrega continua* se refiere a todo lo que sucede después de esa etapa para llevar la nueva versión al campo donde los usuarios finales pueden beneficiarse de ella. La D en CD a veces se expande a “desarrollo” o “implementación” además de “entrega”.

Las tareas principales del CD son probar el código exhaustivamente y cargarlo en las computadoras de los usuarios o en un repositorio de aplicaciones.

En la fase de desarrollo de producto se ejecuta una serie de pruebas para garantizar al máximo que el producto funciona bien. Se puede ejecutar un conjunto más amplio de pruebas de integración, junto con otras pruebas para determinar el impacto en los usuarios, el rendimiento y la seguridad. En una sección posterior de esta lección se describen algunas de estas pruebas.

Las pruebas deben realizarse en sistemas distintos a los de producción. Una falla catastrófica no solo podría hacer caer un sistema, sino que también podría corromper los datos de los usuarios. Además, las pruebas compiten por el tiempo del sistema y degradan el rendimiento para los usuarios.

Por lo tanto, para realizar pruebas, lo mejor es configurar un entorno informático completo que refleje el entorno de producción, con hardware y software similares. El entorno intermedio donde se ejecutan las pruebas antes de la implementación suele denominarse entorno de ensayo.

Por supuesto, no sería práctico hacer que el entorno de prueba tenga el mismo tamaño que el entorno de producción, pero se deberían incluir los elementos esenciales en producción (como las bases de datos).

Un requisito de la automatización de CD es distinguir entre los entornos de prueba y producción. Todas las instalaciones y procesos de código deben adaptarse al entorno de destino.

El CD ofrece el mayor potencial para un desarrollo rápido cuando el código es un servicio que se ejecuta en los propios sistemas de la organización. Si, por ejemplo, se trata de un sitio de venta minorista que ofrece una página web interactiva, se tiene acceso a todos los servidores web. Se

pueden actualizar varias veces al día y revertir los cambios rápidamente si resultan ser un problema para los usuarios.

Herramientas comunes de desarrollo de software

Estas secciones presentan tipos comunes de herramientas utilizadas por los equipos de programación en tres niveles: generación de código, pruebas e implementación.

Compiladores

Una herramienta de programación fundamental es el compilador, que convierte lenguajes de programación muy sofisticados y de alto nivel en instrucciones que se ejecutan en el procesador de la computadora.

Los ordenadores ejecutan *código máquina*, que consiste en cadenas de bits (unos y ceros) que el procesador convierte en instrucciones y datos: cargando un valor de la memoria en un registro, sumando los valores de dos registros, etc. Los procesadores se distinguen por los diferentes formatos y conjuntos de instrucciones que tienen, por lo que también tienen diferentes códigos máquina. Los proveedores intentan inventar nuevos procesadores que utilicen el mismo código máquina para permitir que los clientes ejecuten sus viejos programas en los nuevos procesadores. Cuando los proveedores hacen esto, hablamos de *familias* de procesadores.

La siguiente etapa en los primeros años de la programación fue el lenguaje ensamblador, que proporcionaba términos legibles para humanos, como ADD, para cada instrucción. Los programadores escribían en lenguaje ensamblador y enviaban su código a una herramienta llamada ensamblador para traducir el lenguaje ensamblador a código de máquina. El código de máquina también se denomina lenguaje de máquina.

Luego se crearon lenguajes de nivel superior. Hoy en día, se pueden crear flujos de control complejos sin siquiera especificar sus detalles; basta con indicar los resultados deseados. Estos programas requieren un compilador para convertir el código fuente en código de máquina. Los compiladores pueden realizar transformaciones y optimizaciones bastante inteligentes.

Muchos lenguajes tienen varios compiladores disponibles. Puedes encontrar una selección de compiladores para lenguajes muy populares, como C y Java. En la comunidad de código abierto y gratuito, la mayoría de la gente usa la Colección de compiladores GNU (GCC) o el compilador LLVM para C y C++.

Originalmente, un compilador compilaba cada archivo de código fuente para producir un *archivo de objeto* intermedio y luego combinaba todos los archivos de objeto en un programa invocando otra herramienta llamada *vinculador*. Los compiladores modernos pueden compilar varios

archivos a la vez para realizar optimizaciones que trascienden los límites de los archivos.

Los compiladores solían compilar en lenguaje ensamblador, lo que podía resultar útil porque cada tipo de procesador de ordenador admitía un código de máquina diferente, pero podía admitir el mismo lenguaje ensamblador. Existen múltiples variantes de lenguajes ensambladores. El último paso en la compilación y el enlace era producir código de máquina. Al igual que con el enlace, los compiladores modernos saben cómo ensamblar el código en código de máquina.

Pero en muchos lenguajes modernos existe otra etapa: el código intermedio, que suele denominarse *código byte*. Este código se ha compilado en un formato binario de alto nivel para que sea portable. Por ejemplo, el lenguaje Java se compila en código byte para que el programa pueda cargarse en muchos tipos diferentes de procesadores.

Al producir código de bytes a partir del código fuente, el compilador ha hecho gran parte del trabajo. Luego, cada computadora aloja su propia versión de un programa llamado *máquina virtual* que completa la transformación del código de bytes en un conjunto de instrucciones que la máquina virtual puede ejecutar. A veces, el código de bytes también se compila en código de máquina. Pasar del código de bytes a un conjunto de instrucciones es más conveniente para los usuarios finales que pasar de un lenguaje de programación de alto nivel a código de máquina. Esto fue una gran ventaja para Java cuando se inventó, porque sus diseñadores querían que se ejecutara dentro de un complemento de máquina virtual para navegadores web, donde el procesador y el entorno operativo del usuario podían ser bastante variados.

Los diseñadores de Java promocionaron los beneficios del código de bytes mediante la frase publicitaria “Compila una vez, ejecuta en cualquier lugar”. Más tarde surgieron otras ventajas del código de bytes. Se podían crear nuevos lenguajes que proporcionaban una experiencia muy diferente a los programadores (facilitando el trabajo de programación y produciendo un código más fácil de mantener) y, al mismo tiempo, creando el mismo código de bytes que admitían las máquinas virtuales de Java. Las funciones de estos lenguajes eran fáciles de mezclar en los programas Java existentes.

Por último, existen algunos lenguajes, como Python, para los que no es necesario compilar el código fuente: basta con introducir instrucciones en una herramienta de procesamiento llamada intérprete, que convierte el código directamente en código de máquina y lo ejecuta. Los intérpretes son más lentos que los compiladores, pero algunos se han vuelto lo suficientemente eficientes como para no afectar demasiado al rendimiento. Aun así, algunas bibliotecas populares en el lenguaje Python incluyen funciones a las que los desarrolladores pueden acceder en el lenguaje interpretado, pero que están programadas en el lenguaje C, para acelerar la ejecución.

Muchos lenguajes interpretados también ofrecen compiladores, que generan código de bytes o código de máquina, que luego se ejecuta más rápido que el intérprete.

Existen herramientas de compilación que ayudan a los programadores a gestionar archivos y funciones. Un programa complejo puede contener cientos de archivos y el programador deberá compilar distintas combinaciones de archivos utilizando distintas opciones del compilador en distintos momentos (por ejemplo, para facilitar la depuración). Una herramienta de compilación permite a los programadores almacenar distintas opciones y combinaciones de archivos y elegir fácilmente el tipo de compilación deseado. Maven y Gradle son herramientas de compilación habituales para Java y lenguajes relacionados.

Herramientas de generación de código

El programador no tiene por qué empezar a codificar con una pantalla en blanco. Recientemente han surgido servicios que generan código basándose en la descripción en texto simple de lo que se desea. Se trata de una forma de IA generativa y al igual que otros servicios similares, algunas personas se quejan de que se basa en el trabajo de antiguos programadores sin compensación y produce un código fuente menos sólido (hasta ahora). Aparte de las controversias éticas, muchos programadores dicen que la generación automática de código ha mejorado enormemente su productividad.

Una forma de generación de código que ha estado disponible en muchos lenguajes de programación durante muchos años es la refactorización. Examina un programa grande, que puede evolucionar fácilmente con el tiempo hasta convertirse en una maraña de archivos y funciones. La refactorización mueve las funciones para crear una estructura más lógica para el programa con el objetivo de mejorar la capacidad de mantenimiento y reducir la duplicación del código fuente.

Algunos programadores tienen la tarea de reproducir el funcionamiento de otro código. Es posible que necesiten escribir un programa nuevo para reemplazar un programa heredado con código fuente faltante que debe ser retirado. O pueden estar imitando el programa de un competidor. Este tipo de investigación se llama *ingeniería inversa*. Una herramienta útil para este propósito es un *desensamblador*, que convierte el código de máquina en lenguaje ensamblador. También existen desensambladores para código de bytes.

Depuradores

La mayoría de los programadores pasan más tiempo depurando que codificando. La gente no piensa de forma perfectamente lógica y, por lo tanto, olvida algún detalle que la computadora necesita para ejecutar el programa de la manera deseada. Por lo tanto, es probable que su código falle en el primer intento y puede beneficiarse enormemente de un *depurador* para descubrir el error.

Un depurador respalda esfuerzos de investigación intensivos que pueden reducir horas del

proceso de búsqueda de errores.

Un programador puede pedirle al programa que se detenga en algún punto clave del programa (un *punto de interrupción*), como el comienzo de una función o un bucle. El depurador puede mostrar los valores de las variables e incluso los registros de la computadora en el punto actual del programa. El programador también puede ejecutar cada instrucción, una a la vez, y ver los resultados (*paso a paso*). Si el programador desea ver cuándo y cómo cambia una variable durante la ejecución, puede establecer un *punto de vigilancia*.

El depurador más destacado en el mundo del código abierto y gratuito, en particular para C y C++, es el depurador GNU, que funciona con el compilador GNU mencionado anteriormente. Otros lenguajes también tienen depuradores dedicados.

Herramientas Analíticas

Aunque la depuración suele permitir descubrir errores con bastante rapidez, es mejor eliminar los errores ortográficos y otros problemas básicos en una fase más temprana del proceso de programación. Existen muchas herramientas analíticas ingeniosas para comprobar un programa. El análisis estático examina el código de un programa. El análisis dinámico ejecuta un programa y comprueba si hay problemas durante su ejecución.

Una de las primeras formas de análisis estático se denominaba *linter*. Puede detectar, por ejemplo, si se asigna el valor de una variable de punto flotante a un entero. Esto puede producir problemas, y puede que el compilador lo detecte o no. En producción, puede dar lugar a resultados incorrectos.

Hoy en día, la mayoría de los compiladores pueden hacer el trabajo de un linter. Algunos compiladores, como el del lenguaje Rust, se destacan por su rigurosidad al rechazar código mal escrito.

Muchos otros tipos de herramientas analíticas se ejecutan por separado del compilador. Por ejemplo, las herramientas de análisis de seguridad pueden encontrar problemas que hagan que un programa sea vulnerable a la piratería. Un error común de los desarrolladores, por ejemplo, es cuando su código llama a una función y no verifica si esa función devolvió un error.

Entornos de desarrollo integrados (IDE)

Muchos programadores utilizan editores de texto para introducir y editar su código. Los editores de texto son diferentes de los procesadores de texto, que introducen una gran cantidad de formato (como cursiva y negrita, viñetas y listas numeradas, etc.). El procesador de texto produce texto sin adornos, algo que requieren los lenguajes de programación.

También existen herramientas sofisticadas dedicadas a ayudar a los programadores a desarrollar sus programas. Estas herramientas están alertas al lenguaje de programación en uso. Por ejemplo, si comienza a escribir el nombre de una variable de función, la herramienta puede sugerirle que complete la tarea. Estas herramientas pueden verificar errores mientras está codificando, formatear el código de una manera consistente y agradable, ejecutar herramientas analíticas y depuradores, compilar el código, manejar los check-ins en los sistemas de control de versiones y encargarse de otras tareas por usted. Por lo tanto, se denominan *entornos de desarrollo integrados* (IDE).

Eclipse es un IDE de código abierto popular.

Tipos comunes de pruebas de software

Las pruebas son una parte importante del desarrollo de software. Los programadores ejecutan pruebas unitarias a medida que desarrollan el código. Otros tipos de pruebas se ejecutan normalmente cuando un programador devuelve el código al repositorio principal o durante la implementación.

Pruebas unitarias

Hemos visto que un programador debe tener mucho cuidado para encontrar errores antes de enviar el código para su integración en el repositorio principal. Las pruebas unitarias son fundamentales para detectar errores.

Escribir estas pruebas es tanto un arte como una ciencia, y el volumen de código de prueba puede superar el volumen de código de producción. Incluso existe un modelo de desarrollo llamado *desarrollo impulsado por pruebas* (TDD), en el que los programadores escriben pruebas antes de escribir el código que quieren probar. Los defensores del TDD afirman que cubre las lagunas en las pruebas y ayuda a garantizar que el código haga lo que el programador quiere que haga.

Es importante comprobar si hay errores durante la ejecución del programa, así como también si hay errores. Si el usuario u otra parte del programa envía una entrada no válida a una función, es importante que la función detecte el problema y envíe un mensaje de error adecuado.

JUnit es una popular herramienta de código abierto para ejecutar pruebas unitarias en programas Java.

Integración, regresión y pruebas de humo

Si bien las pruebas unitarias se centran en las acciones individuales de funciones específicas, el equipo también debe ejecutar pruebas a un nivel superior para asegurarse de que el producto, en

su conjunto, funcione correctamente. Las pruebas generalmente imitan el comportamiento del usuario. Por ejemplo, en una aplicación de gestión de restaurantes, las pruebas podrían verificar si el usuario recibe el artículo que solicitó.

Cuando un cambio en un programa rompe alguna función que funcionaba antes, la falla se llama *regresión* y las pruebas se llaman *pruebas de regresión*.

Mientras un equipo prepara un producto para su lanzamiento, la primera etapa de pruebas suele ser muy breve. El equipo comprueba las actividades más importantes que realiza un programa y detiene las pruebas si surgen errores, ahorrando así tiempo. Este tipo de pruebas se denominan pruebas de humo, porque una aplicación que funciona mal con tanta facilidad es como un dispositivo defectuoso que se incendia.

Algunos productos implican la interacción del usuario; las aplicaciones web y móviles son ejemplos comunes. Por lo tanto, se han creado herramientas para emular las interacciones del usuario. La prueba se ejecuta automáticamente y activa el código que se habría ejecutado si un usuario presionara el botón. Selenium es una herramienta popular en esta categoría.

Pruebas de aceptación

Los programas con una interfaz de usuario necesitan un nivel adicional de pruebas, además de demostrar que reaccionan correctamente a determinadas entradas. Los programas también deben verse bien en la pantalla. Las pruebas de aceptación verifican el impacto del programa en el usuario. Los equipos de control de calidad generalmente realizan estas pruebas después de que las pruebas de integración y regresión demuestren que el programa es formalmente correcto y cumple con las expectativas del usuario.

Pruebas de seguridad

Obviamente, la seguridad es fundamental, e incluso un pequeño fallo de seguridad puede exponer a una organización a daños importantes. Hemos visto que los programadores pueden ejecutar herramientas analíticas para comprobar la seguridad del código. En la etapa de control de calidad, las pruebas también pueden determinar si el programa tiene vulnerabilidades. Las pruebas ejecutan el programa con entradas maliciosas y se aseguran de que el programa rechace la entrada sin fallar, realizar acciones no deseadas o revelar información confidencial.

Un tipo de prueba que ha demostrado ser útil en algunas situaciones es la prueba de fuzzing. El marco de prueba simplemente genera cadenas de basura aleatoria y las envía como entrada al programa. Esto puede parecer una pérdida de tiempo, pero a menudo descubre vulnerabilidades que las pruebas ordinarias no detectan.

Pruebas de rendimiento

Una vez que otras pruebas determinan que el programa funciona correctamente, los equipos deben determinar si se ejecuta con la suficiente rapidez. Las pruebas de rendimiento requieren un entorno similar a aquellos en los que los usuarios interactuarán con el programa. Por ejemplo, si los usuarios enviarán solicitudes desde una gran distancia a través de una red, las pruebas de rendimiento también deben realizarse en una red de larga distancia.

Algunas bibliotecas de programación se prueban a través de *benchmarks*: pruebas estándar utilizadas para comparar diferentes bibliotecas o diferentes versiones de la misma biblioteca.

Entornos de implementación comunes

Una herramienta de CI/CD permite crear canales de forma sofisticada. Al igual que los programas informáticos, un canal de producción puede contener pruebas y ramificaciones. Mediante la ramificación, puede realizar un conjunto de actividades en el entorno de prueba y otro en el entorno de producción. Puede utilizar la herramienta para instalar automáticamente la base de datos correcta u otro software necesario para diferentes programas.

La CD se superpone con DevOps. En entornos de nube, las herramientas de CD y DevOps crean los sistemas informáticos virtuales de forma automatizada con todos los componentes necesarios para que se ejecuten los programas. Las herramientas automatizadas (a veces llamadas herramientas de orquestación) comprueban las fallas de los sistemas virtuales y los reinician automáticamente.

La función principal de la herramienta CD es iniciar y recorrer cada canal de procesamiento. La herramienta verifica los resultados de cada etapa del canal de procesamiento y decide si continuar o detenerse. La herramienta también permite programar y registrar sus actividades.

A menudo, es necesario ejecutar una tarea repetidamente con pequeñas variaciones. Por ejemplo, los equipos distinguen entre implementar en un entorno de prueba y en producción. Por lo tanto, las herramientas de CD proporcionan parámetros que se pueden completar con diferentes valores cuando se ejecuta la canalización.

A veces, un proceso requiere comandos que tradicionalmente se ingresaban en la terminal. Por lo tanto, una herramienta de CD proporciona mecanismos para ejecutar comandos arbitrarios. Por lo general, proporciona ganchos (hooks) como `preprocess` para ejecutar comandos antes de una etapa del pipeline y `postprocess` para ejecutar comandos después de una etapa del pipeline.

Jenkins es probablemente la herramienta de código abierto más popular para la orquestación descrita en esta sección.

Ejercicios guiados

1. ¿Cuáles son algunas formas de comprobar la seguridad de un programa?

2. ¿Por qué escribirías múltiples pruebas unitarias para una sola función del programa?

Ejercicios exploratorios

1. Su equipo ha heredado una aplicación antigua que se ejecuta con lentitud y a la que es difícil agregarle funciones. ¿Cuáles son algunas formas de mejorar la aplicación sin tener que descartarla y escribir una nueva desde cero?

2. En proyectos grandes, con frecuencia, el Equipo A quiere que se implemente una característica en una parte del sistema que mantiene el Equipo B, pero el Equipo B no considera que la característica sea una prioridad. ¿Cómo puede el Equipo A codificar la característica como parte del proyecto del Equipo B?

Resumen

En esta lección se han explicado los tipos de herramientas que se utilizan en el proceso de desarrollo: compiladores y otras herramientas de generación de código, analizadores, pruebas y herramientas de CI/CD que automatizan la integración y la entrega. Existen muchas opciones para cada una de estas actividades, y una herramienta que hoy es popular puede ser reemplazada en un año. Comprender cómo se combinan todas estas herramientas en el proceso de desarrollo le ayudará a identificar lo que necesita.

Respuestas a ejercicios guiados

1. ¿Cuáles son algunas formas de comprobar la seguridad de un programa?

   Primero, los expertos humanos pueden examinar el código.

   Existen muchas herramientas de análisis estático y dinámico para detectar malas prácticas de programación que exponen un programa a amenazas de seguridad.

   Otras herramientas envían información maliciosa a los programas en ejecución y verifican sus reacciones.

2. ¿Por qué escribiría varias pruebas unitarias para una sola función de programa?
   Una función de programa normalmente tiene que ejecutarse en muchas variedades de entrada, y cada variedad merece su propia prueba. Por ejemplo, la función podría manejar un valor de entrada de cero de una manera especial. También necesita anticipar la entrada no válida y escribir pruebas para demostrar que la función la maneja adecuadamente.

Respuestas a Ejercicios Exploracionales

1. Su equipo ha heredado una aplicación antigua que se ejecuta lentamente y a la que es difícil agregarle funciones. ¿Cuáles son algunas formas de mejorar la aplicación, sin descartarla y escribir una nueva desde cero?

   Primero, asegúrese de que el proyecto esté bajo control de versiones, si no estaba allí ya.

   Después de agregar una nueva función, ejecute pruebas de regresión para determinar dónde falla el programa y asigne programadores para que averigüen qué funciones son responsables. Estas funciones se pueden reemplazar de forma selectiva.

   Las pruebas de rendimiento pueden identificar funciones particulares que se ejecutan lentamente, de modo que pueda concentrar sus esfuerzos en corregir o reemplazar el código más ineficiente.

   Si el código está en un lenguaje que ya no es popular, considere agregar funciones en un lenguaje preferido por el equipo. Asegúrese de que las funciones en el nuevo lenguaje se puedan integrar con las funciones antiguas.

2. En proyectos grandes, con frecuencia, el Equipo A quiere que se implemente una característica en una parte del sistema que mantiene el Equipo B, pero el Equipo B no ve la característica como una prioridad. ¿Cómo puede el Equipo A codificar la característica como parte del proyecto del Equipo B?

   El Equipo B puede permitir que el Equipo A cree una nueva rama, codifique la característica y envíe la rama al Equipo B para que se fusione con su proyecto. Sin embargo, el Equipo A no debe tener la autoridad para hacer lo que quiera. El Equipo B debe proporcionar documentación y ayuda para que el Equipo A siga los estándares del proyecto. Un miembro del Equipo B también debe revisar la presentación del Equipo A y ejecutar todas las formas habituales de pruebas de integración.

   Esta forma de colaboración a veces se llama InnerSource, porque se parece al código abierto pero se lleva a cabo dentro de una sola organización.

056.2 Gestión de código fuente

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 056.2

Peso

3

Áreas de conocimiento clave

- Comprender los repositorios de código fuente (públicos y privados)
- Comprender los principios de la gestión del código fuente y la organización del repositorio
- Conocimiento de los sistemas SCM comunes (Git, Subversion, CVS)
- Conocimiento de los términos Sistema de control de versiones (VCS), Sistema de control de revisiones y Sistemas de gestión de código fuente (SCM)

Lista parcial de archivos, términos y utilidades

- Repositorios de código fuente
- Confirmaciones, ramas y etiquetas
- Ramas de características, desarrollo y lanzamiento.
- Subrepositorios
- Fusiones de código

Linux Professional Institute

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	056 Colaboración y comunicación
Objectivo:	056.2 Gestión del código fuente
Lección:	1 de 1

Introducción

Cualquiera que haya editado alguna vez un documento de texto en equipo conoce los problemas que conlleva este tipo de colaboración: ¿cuál es la versión actual? ¿Dónde está guardada esta versión? ¿Alguien la está editando actualmente? ¿Quién ha realizado qué comentarios o cambios en el texto, cuándo y por qué? El resultado suele ser que existen diferentes versiones del documento y, en el peor de los casos, una colección de versiones que nadie conoce.

Imaginemos ahora un proyecto de software con cientos de archivos en el que trabajan desarrolladores de todo el mundo desarrollando nuevas funciones, solucionando errores, dividiendo partes y desarrollándolas por separado, etc. Un proceso de desarrollo de este tipo ya no es posible sin las herramientas adecuadas.

Un software especial para la gestión del código fuente (SCM), también conocido como sistema de control de versiones (VCS) o sistema de control de revisiones (RCS), ofrece una solución a este problema, ya que elimina los problemas que acabamos de describir.

En el mundo del desarrollo de software, SCM es un pilar fundamental que salvaguarda la integridad de su código fuente. Imagínelo como un guardián diligente que rastrea

meticulosamente cada modificación y cambio que se le hace a su código fuente a lo largo del tiempo.

Sistema de gestión y repositorio de código fuente

El sistema de gestión de código fuente es el corazón de un proyecto de software. Aunque se realizan trabajos importantes en foros de debate y otros lugares, el sistema SCM representa tanto la historia del proyecto como su estado actual como entidad viva.

El repositorio de código fuente es una especie de taller digital para un proyecto. Así como un taller físico almacena todas las herramientas y elementos necesarios para fabricar una pieza de trabajo, un repositorio de código fuente almacena todos los archivos, documentos y códigos relacionados con un proyecto de software. Proporciona un entorno estructurado para organizar y gestionar los activos del proyecto.

La información se almacena normalmente en forma de árbol de directorios. Los sistemas SCM suelen utilizar un modelo cliente-servidor, en el que cualquier usuario puede extraer datos del repositorio y también introducirlos en este. El sistema lleva un registro de quién realizó cada cambio y cuándo se realizó, lo que garantiza la transparencia y la rendición de cuentas dentro del equipo.

Imagina que estás trabajando en un proyecto con varios desarrolladores y se descubre un error en el código. Con un SCM, puedes examinar fácilmente los cambios entre versiones para identificar el cambio de código específico responsable del error.

Como el sistema recuerda cada versión de los archivos a medida que cambian, un usuario tiene acceso a cualquiera de estas versiones y puede volver a versiones anteriores (en caso de que se hayan realizado cambios incorrectos). Por lo tanto, el repositorio también es una especie de archivo, que otorga acceso a todos los cambios realizados en el proyecto y al estado del proyecto en cualquier momento de su historia.

Los sistemas SCM ahorran espacio al realizar un seguimiento de los cambios realizados en un archivo en lugar de almacenar el archivo completo cada vez que se realiza un cambio. Este eficiente método de almacenamiento garantiza que las versiones históricas sean accesibles sin consumir recursos excesivos.

Muchas herramientas, como la integración continua/entrega continua (CI/CD) y las pruebas, giran en torno al sistema SCM. Las personas también construyen su reputación a través del sistema al hacer que sus contribuciones sean visibles para todos. Por lo tanto, la gestión del código fuente no se trata solo de realizar un seguimiento de los cambios; se trata de preservar la integridad de su base de código y fomentar la colaboración entre los desarrolladores, lo que garantiza que sus

proyectos puedan evolucionar en un panorama dinámico y en constante cambio.

Los sistemas SCM más populares son Git, Subversion y CVS. Al igual que muchas otras funciones de software, SCM suele ser proporcionado por proveedores especializados. En otras palabras, es un software como servicio (SaaS) o un servicio "en la nube": los participantes tienen el software en sus sistemas locales para gestionar sus cambios personales, pero cargan sus cambios en un repositorio central que se beneficia de las características típicas de la nube, como tiempo de actividad 24 horas al día, 7 días a la semana, copias de seguridad y acceso seguro.

Millones de desarrolladores utilizan ahora servicios en la nube, en particular GitHub. GitLab es una alternativa basada en código abierto. Tanto GitHub como GitLab permiten a las personas trabajar en los repositorios en la nube del proveedor o configurar una versión local del sistema SCM. Una ventaja de trabajar en esos sistemas es la reputación que se puede ganar a través de sus sistemas de "estrellas", que permiten a los usuarios calificar el trabajo de los demás. Los servicios en la nube añaden atracciones adicionales, como rastreadores de solicitudes de cambio, sistemas de calificación, wikis y foros de debate.

Los repositorios pueden contener tanto proyectos personales como corporativos, lo que significa que no son necesariamente exclusivos del uso empresarial. Cualquier desarrollador que quiera iniciar un proyecto de desarrollo o trabajar en un proyecto de código abierto, copiándolo y modificándolo según sus necesidades, puede hacerlo. En todos estos casos, es importante tener un control firme sobre quién puede acceder a su repositorio.

Comprender la terminología relacionada con el control de versiones y la gestión del código fuente es fundamental para comprender su uso. En las siguientes secciones, analizamos en profundidad algunos de estos conceptos y términos.

Confirmaciones, etiquetas y ramas

Un desarrollador crea una *confirmación (commit)* cada vez que hace un cambio en el repositorio. Las confirmaciones representan instantáneas de los cambios realizados en el código base en un momento determinado. Cada confirmación incluye metadatos como el nombre del autor, una marca de tiempo y un mensaje descriptivo que explica los cambios. Las confirmaciones ayudan a los desarrolladores a realizar un seguimiento de la evolución del código base y comprender el historial de cambios específicos.

Considere un equipo de desarrolladores que trabaja en una aplicación web. Cada vez que realizan cambios en el código base, crean una confirmación para documentar esos cambios. Por ejemplo, alguien que agrega una nueva característica a la aplicación puede crear una confirmación con un mensaje como "Se agregó una característica de autenticación de usuario". Esta confirmación captura el estado del código base después de que se implementó la característica.

Cuando un desarrollador corrige un error, el mensaje de confirmación generalmente se refiere al número del error en el sistema de seguimiento de errores del proyecto.

Las *etiquetas (Tags)* son referencias con nombre a confirmaciones específicas. Por lo general, marcan puntos importantes en el historial del proyecto, como lanzamientos o hitos. Las etiquetas proporcionan una forma de etiquetar y hacer referencia a versiones importantes del código base, lo que facilita la gestión y la navegación por el historial del proyecto.

Las *ramas (branches)* entran en juego cuando los desarrolladores necesitan trabajar en diferentes tareas al mismo tiempo. Las ramas son líneas de desarrollo independientes que se apartan del código base principal. Permiten a los desarrolladores trabajar en funciones o correcciones de forma aislada sin afectar el código principal hasta que estén listos para la integración. Las ramas ayudan a organizar los esfuerzos de desarrollo y facilitan la colaboración entre los miembros del equipo.

Por ejemplo, un desarrollador está trabajando en agregar una nueva característica a la aplicación mientras otro está solucionando un error. Cada uno puede crear ramas separadas para aislar sus cambios. Una vez que su trabajo esté completo, pueden *fusionar (merge)* sus ramas nuevamente en la base de código principal (<<branches> >).

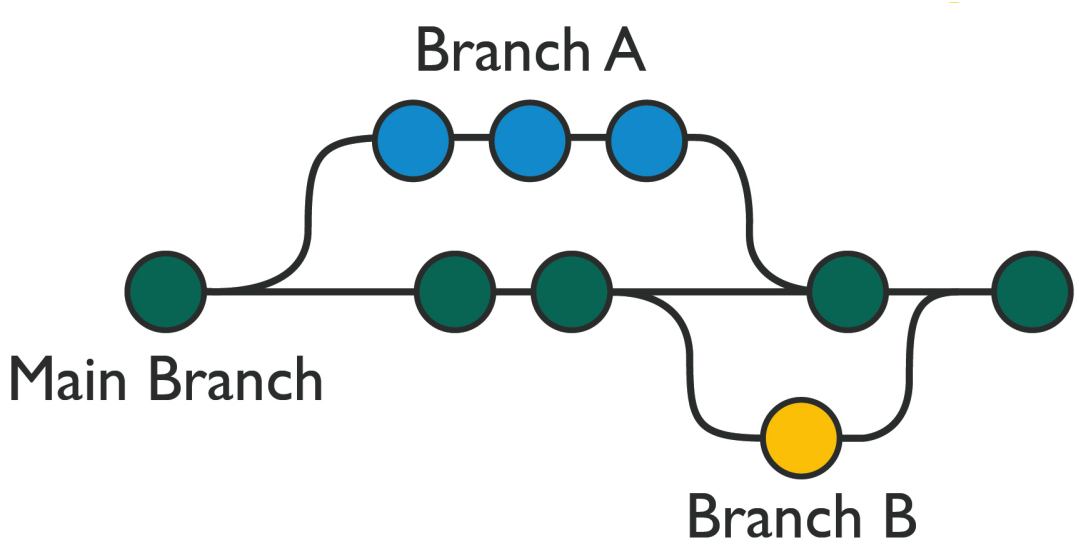

Figure 16. Branches

Cuando varias personas trabajan en el mismo archivo o están enviandolos en otra rama, en ocasiones puede suceder que dos personas realizan un cambio en la misma línea de ese archivo. Cuando la segunda persona que realiza un cambio intenta despacharlo, el sistema advierte de un *conflicto*.

Algunos VCS simplemente no permiten que un desarrollador registre un archivo si contiene un

conflicto con la versión actual en el repositorio. El desarrollador debe verificar la versión actual y averiguar cómo resolver el conflicto, y luego registrar una nueva versión.

Los sistemas basados en la nube ofrecen solicitudes de fusión o de extracción. Estas solicitudes las crea un desarrollador que ha trabajado en un sistema o rama local y cree que sus cambios son adecuados para incluirlos en otra rama. Los desarrolladores de la rama de destino deciden si aceptan la solicitud.

Subrepositorios

A menudo sucede que para el desarrollo de un proyecto de software (por ejemplo, un sitio web complejo) se necesita el código de otro proyecto independiente (por ejemplo, un reproductor multimedia). En lugar de copiar el código del reproductor multimedia en parte o en su totalidad en su propio proyecto, muchos VCS ofrecen la función de *subrepositorios*, también conocidos como *submódulos*.

En nuestro ejemplo, el repositorio del reproductor multimedia se integra en el repositorio del sitio web como subrepositorio. Aparece como un directorio independiente en el árbol de directorios. Esto significa que la base de código del reproductor multimedia está completamente disponible, pero sigue siendo independiente. Si es necesario, incluso se puede actualizar desde el repositorio original del reproductor multimedia. Esta capacidad resulta valiosa para gestionar proyectos complejos con numerosas dependencias o para integrar bibliotecas y marcos de terceros en una base de código. Los submódulos mejoran la organización del proyecto y facilitan la colaboración al permitir que los desarrolladores trabajen con bases de código interconectadas de manera eficiente.

Uso general de un sistema de gestión de control de fuentes

Cada participante de un proyecto comienza creando una identidad, normalmente vinculada a su dirección de correo electrónico única. Un sistema basado en la nube administra las identidades a través de cuentas, como lo hacen los sitios de redes sociales y otras organizaciones.

Los nuevos desarrolladores pasan por una secuencia como la siguiente cuando utilizan un sistema SCM:

1. Instale el software SCM, si aún no está incluido en su sistema operativo.
2. Obtener todo el proyecto en el sistema local de una sola vez, también conocido como *clonación*.
3. A partir de este paso, trabaje localmente (en una o más ramas, si es necesario) y envíe los cambios al repositorio.
4. Extraiga la versión reciente del repositorio antes de la próxima sesión de trabajo.

Los administradores de proyectos deciden en quién confiar y en quién dar acceso al repositorio. Los desarrolladores senior tienen la importante responsabilidad de decidir cuándo los cambios enviados por otros colaboradores están listos para incluirse en la rama principal del repositorio.

En los sistemas basados en la nube, el propietario de un proyecto puede controlar la accesibilidad de un repositorio configurando su visibilidad como pública o privada. Los repositorios públicos otorgan acceso de lectura a cualquier persona en Internet. Sin embargo, los repositorios privados limitan el acceso únicamente al propietario, a las personas con las que el propietario ha compartido explícitamente el acceso y, en el caso de los repositorios de la organización, a miembros específicos de la organización.

Imaginemos un equipo de desarrolladores que trabaja en un sitio web de comercio electrónico. Deciden añadir una nueva función que permita a los clientes realizar un seguimiento de sus pedidos. Para implementar esta función, crean una rama de funciones con un nombre como `seguimiento de pedidos (order-tracking)`, donde pueden trabajar en los cambios de código necesarios sin afectar al código base principal. Una vez que la función está completa y probada, fusionan esta rama con la rama de desarrollo principal para una mayor integración y prueba.

La rama de desarrollo principal funciona como un punto central donde se reúnen todas las nuevas funciones para realizar pruebas. Por ejemplo, si varios desarrolladores trabajan en distintas funciones al mismo tiempo, pueden fusionar sus ramas de funciones en la rama de desarrollo para garantizar que todo funcione en conjunto sin problemas. Este proceso de integración ayuda a identificar y resolver cualquier conflicto o problema de compatibilidad desde el principio.

Cuando llega el momento de lanzar una nueva versión del sitio web de comercio electrónico, el equipo crea una rama de lanzamiento, como `v2.0`, a partir de la rama de desarrollo. Se concentran en estabilizar la base de código, corregir errores de último momento y realizar pruebas exhaustivas para garantizar un lanzamiento sin problemas. Una vez que el lanzamiento está listo, el código de la rama de lanzamiento se implementa en producción y el ciclo comienza de nuevo.

Sistemas comunes de control de versiones

Algunos de los sistemas de control de versiones más conocidos son Git, Subversion (también conocido como SVN) y CVS. Todos ellos son de código abierto.

Git es un sistema de control de versiones distribuido que se utiliza ampliamente en el desarrollo de software y en otros campos. Al utilizar Git, cada desarrollador tiene una copia completa del código base en su computadora.

Este enfoque descentralizado permite a los desarrolladores trabajar sin conexión y colaborar sin problemas, sin depender de un servidor central. Es decir, los desarrolladores pueden trabajar de forma independiente en diferentes funciones o correcciones y combinar sus cambios sin problemas. Incluso si el servidor central se desconecta, los desarrolladores pueden seguir trabajando y compartir actualizaciones entre ellos.

Pensemos en el desarrollo del núcleo Linux, que inicialmente dependía de un sistema de control de versiones centralizado llamado BitKeeper. Cuando se revocó el estatus gratuito de BitKeeper, Linus Torvalds y la comunidad Linux desarrollaron Git como una alternativa distribuida. Esta decisión permitió el desarrollo no lineal y la gestión eficiente de proyectos grandes como el núcleo Linux. El éxito de Git para el núcleo Linux (un proyecto extremadamente complejo con miles de desarrolladores e innumerables ramas) muestra el poder y la escalabilidad de Git.

Hoy en día, la mayor parte del desarrollo de software utiliza Git. Las ofertas de SaaS extremadamente populares se basan en este.

Git trata los conflictos entregando al desarrollador un archivo con ambas versiones de la línea modificada, y marcando claramente de qué versión del archivo provienen las líneas. El desarrollador debe decidir cómo resolver el conflicto y registrar una versión coherente del archivo.

Subversion (SVN) probablemente fue el sistema de gestión de contenido más popular antes de que se inventara Git. A diferencia de Git, Subversion está centralizado: el historial de versiones reside en un servidor central. Los desarrolladores se conectan a este servidor para realizar cambios, lo que garantiza que todos trabajen con la última versión del código base.

Supongamos que forma parte de un equipo que trabaja en un proyecto que utiliza Subversion. Cada vez que necesita realizar cambios en el código base, se conecta al servidor SVN central para extraer una copia funcional del código. Esto garantiza que está trabajando con la versión más actualizada del proyecto. Después de realizar los cambios, los envía de nuevo al servidor y actualiza el repositorio central con las modificaciones. El flujo de trabajo centralizado ayuda a mantener la coherencia y garantiza que todos trabajen para alcanzar los mismos objetivos.

Antes de Subversion, *CVS* era un sistema de control de versiones centralizado muy popular. Tenía problemas de diseño que llevaron al diseño de Subversion como alternativa.

Ejercicios guiados

1. Mencione tres características principales de los sistemas SCM.

2. Describa el concepto de etiquetado en los sistemas SCM y explique por qué es importante para gestionar los lanzamientos de software.

¿Cuál es la diferencia entre una rama y un subrepositorio en un sistema SCM?

+

Ejercicios exploratorios

1. Compare Git y Subversion (SVN) en términos de su arquitectura y flujo de trabajo.

2. ¿Qué es el «índice» o «área de preparación» en Git?

3. Describa la estrategia de ramificación basada en el tronco de Git.

4. ¿Cuáles de los siguientes sistemas SCM son de código abierto?

Git	
Mercurial	
Subversion	
GitHub	
Bitbucket	
GitLab	

Resumen

En esta lección se explicó el papel central de los sistemas de gestión de código fuente en el desarrollo de software moderno. Aprendió los términos básicos y las formas de utilizar el sistema, incluidos el repositorio, las ramas, las etiquetas y las fusiones.

Respuestas a ejercicios guiados

1. Mencione tres características principales de los sistemas SCM.
   - Registro de cambios en el código fuente
   - Gestión del acceso (simultáneo) al código fuente por parte de los desarrolladores
   - Capacidad de restaurar cualquier estado de desarrollo de los archivos o de todo el proyecto
2. Describa el concepto de etiquetado en los sistemas SCM y explique por qué es importante para gestionar las versiones de software.

   El etiquetado es la práctica de asignar etiquetas o nombres descriptivos a confirmaciones específicas dentro de la base de código, que sirven como marcadores para puntos significativos en la historia del proyecto, como versiones o hitos. Estas etiquetas ofrecen un medio conveniente para hacer referencia a versiones particulares de la base de código y monitorear la evolución del proyecto a lo largo del tiempo.

3. ¿Cuál es la diferencia entre una rama y un subrepositorio en un sistema SCM?

   Una rama es una línea de desarrollo paralela en un proyecto, como para corregir errores o desarrollar nuevas características, que generalmente se fusiona nuevamente con la rama de desarrollo principal tan pronto como se completa la tarea. Un subrepositorio o módulo de suma es un proyecto independiente cuyo repositorio se integra en un proyecto para acceder a su base de código. El subrepositorio aparece como un directorio en el árbol de directorios del proyecto y permanece independiente.

Respuestas a Ejercicios Exploracionales

1. Compare Git y Subversion (SVN) en términos de su arquitectura y flujo de trabajo.

   Git es un sistema de control de versiones distribuido (DVCS), que permite a cada desarrollador tener una copia completa del código base y trabajar incluso sin conexión en el código fuente. Git ha ganado una gran popularidad entre los desarrolladores debido a su velocidad, flexibilidad y sólidas capacidades de ramificación y fusión. SVN es un VCS centralizado, con el historial de versiones que reside en un servidor central. Sigue siendo popular en ciertos entornos empresariales debido a su naturaleza centralizada y su conjunto de características maduras.

2. ¿Qué es el “índice” o “área de preparación” en Git?

   El índice o área de preparación es una capa intermedia entre la copia de trabajo local del proyecto y la versión actual en el servidor. Es un archivo en el que se almacena toda la información para el próximo commit de un usuario.

3. Describa la estrategia de ramificación basada en el tronco de Git.

   El desarrollo basado en el tronco es una estrategia que enfatiza la integración frecuente de cambios en la base de código principal (tronco). Los desarrolladores trabajan en ramas de características de corta duración y las fusionan en el tronco varias veces al día, lo que garantiza una integración continua y una retroalimentación rápida.

4. ¿Cuáles de los siguientes sistemas SCM son de código abierto?

Git	X
Mercurial	X
Subversion	X
GitHub	
Bitbucket	
GitLab	X

056.3 Herramientas de comunicación y colaboración

Referencia al objetivo del LPI

Open Source Essentials version 1.0, Exam 050, Objective 056.3

Peso

2

Áreas de conocimiento clave

- Comprender las herramientas comunes para la comunicación.
- Comprender las formas comunes de capturar y proteger el conocimiento
- Comprender las herramientas comunes para la gestión y publicación de la información.
- Comprender los tipos comunes de documentación
- Comprender las funciones de colaboración comunes de las plataformas de gestión de código fuente
- Comprender los conceptos de plataformas y aplicaciones administradas independientes, federadas y centralizadas.

Lista parcial de archivos, términos y utilidades

- Mensajería instantánea
- Plataformas de chat
- Listas de correo
- Boletines
- Rastreadores de problemas y rastreadores de errores
- Informes de errores
- Solicitudes de fusión y solicitudes de extracción

- Servicio de asistencia técnica y sistemas de tickets
- Wikis
- Sistemas de gestión documental (DMS)
- Sitios web de documentación
- Sitios web de productos
- Sistemas de gestión de contenidos (CMS)
- Documentación de arquitectura
- Documentación del usuario
- Documentación del administrador
- Documentación del desarrollador

Linux Professional Institute

Lección 1

Certificación:	Open Source Essentials
Versión:	1.0
Tema:	056 Colaboración y comunicación
Objectivo:	056.3 Herramientas de comunicación y colaboración
Lección:	1 de 1

Introducción

Muchos proyectos de código abierto cuentan con participantes activos en todo el mundo. La colaboración se produce principalmente en un “espacio virtual” y las personas involucradas suelen estar distribuidas en distintos países, continentes y zonas horarias. Además, suelen hablar distintos idiomas nativos. Esto significa que puedes estar en cualquier parte del mundo para hacer una contribución a un proyecto de código abierto, ¡sin importar tu país o idioma!

Si bien esta diversidad hace que contribuir a un proyecto de código abierto sea extremadamente gratificante, ya que permite ampliar el alcance y aprender mucho, al mismo tiempo puede hacer que la comunicación y la coordinación sean todo un desafío. Una comunicación eficiente y eficaz es clave para el éxito y la sostenibilidad de cualquier proyecto de código abierto. Para garantizar una buena comunicación, los proyectos de código abierto han creado estructuras y utilizan una variedad de herramientas que ayudan a facilitar las contribuciones y la cooperación sea más eficaz.

Otro desafío es que los proyectos de código abierto, a menudo impulsados principalmente por voluntarios, enfrentan cierta fluctuación en la participación. Las vidas y los pasatiempos de las

personas cambian, y algunas pueden perder el interés o simplemente quedarse sin tiempo. Puede contribuir a un proyecto de código abierto durante años, pero cuando cambia de trabajo, se casa o comienza a criar hijos, es posible que ya no tenga suficiente tiempo para hacer voluntariado.

Por lo tanto, además de proporcionar medios de comunicación eficientes a través de herramientas adecuadas, los proyectos de código abierto deben garantizar la preservación y el intercambio de conocimientos para evitar "reinventar la rueda". La preservación de la información también ayuda a los contribuyentes a aprender de los errores pasados de otros contribuyentes y evitar cometer los mismos errores.

Esta lección presenta herramientas comunes para cooperar en un proyecto de código abierto y le presenta formas de comunicarse en una comunidad internacional. ¡Descubrirá que existe una manera fácil para que todos hagan su primera contribución!

Como ejemplo de comunicación e intercambio de información, analizaremos LibreOffice. LibreOffice es una suite de productividad de oficina de código abierto que está disponible en más de cien idiomas. Sus usuarios finales van desde el usuario doméstico ocasional hasta los grandes gobiernos y corporaciones. Asimismo, hay una amplia gama de colaboradores: no solo desarrolladores, sino también localizadores, autores de documentación, personal de marketing, ingenieros de control de calidad, administradores de infraestructura, diseñadores gráficos, UX, y muchos más.

En otras palabras: en LibreOffice, como en muchos otros proyectos de código abierto, puedes aportar tus habilidades y talentos en el área en la que te sientas cómodo. No es necesario que seas una persona con formación técnica o un desarrollador: puedes aportar tu creatividad y talento artístico o también tus habilidades lingüísticas. El proyecto también ha lanzado un sitio web que muestra las distintas áreas de contribución: https://whatcanidoforlibreoffice.org. De modo que LibreOffice ilustrará muy bien muchos aspectos del trabajo comunitario.

Formas de comunicarse

Antes de analizar los detalles de las herramientas de comunicación, es importante entender cómo funciona la comunicación en general, ya que estas consideraciones influyen en la elección de herramientas.

Los proyectos de código abierto se comunican de dos formas diferentes. En la comunicación sincrónica, las personas se comunican al mismo tiempo. Por ejemplo, las conversaciones directas, pero también las videoconferencias o las llamadas telefónicas. En la comunicación asincrónica, las personas se comunican en momentos diferentes. Por ejemplo, el correo electrónico y los SMS, pero también una carta postal o un fax.

Sin embargo, esta distinción no siempre es fácil de hacer. Por ejemplo, una aplicación de mensajería en un teléfono móvil como WhatsApp, Telegram, Signal o Element es técnicamente una forma asincrónica de comunicación. Sin embargo, si ambos interlocutores están en línea al mismo tiempo y responden al instante, entran en una conversación directa.

Este ejemplo muestra que una parte de la comunicación también depende de cómo las personas utilizan sus herramientas y qué expectativas tienen. Cada tarea puede requerir un conjunto diferente de herramientas de comunicación. Las siguientes secciones explicarán estas ideas en detalle.

Comunicación sincrónica

La comunicación sincrónica es una forma de comunicación muy eficaz, pero también muy exigente. Reúne a personas al mismo tiempo en el mismo espacio físico o virtual.

Una reunión directa es ideal para hablar de las cosas de forma interactiva en lugar de enviar mensajes de correo electrónico largos que conllevan el riesgo de malentendidos. Imagina que quieres aprender más sobre un proyecto de código abierto y conocer a la gente de la comunidad. Los mensajes de correo electrónico y los sitios web pueden ayudar a hacer una presentación y tener una barrera de entrada más baja, pero una primera impresión verdaderamente significativa se produce cuando realmente puedes hablar con alguien en persona; son estas interacciones directas las que generalmente fascinan a las personas sobre un proyecto de código abierto y las hacen querer contribuir.

Además de conocerse mejor, las reuniones sincrónicas también son excelentes para hablar de las cosas de forma interactiva, por ejemplo, en caso de conflictos o problemas, o cuando hay que compartir un mensaje negativo.

La desventaja es que los proyectos internacionales enfrentan un desafío que la tecnología no puede superar: las zonas horarias. Si tienes colaboradores en diferentes continentes, será muy difícil encontrar horarios de reunión adecuados para todos. Es posible que alguien en Australia se esté despertando cuando alguien en Europa está a punto de terminar su día. Como desafío adicional, algunas personas pueden trabajar solo por la noche o los fines de semana, mientras que otras prefieren el horario de oficina durante los días hábiles.

Además, el inglés no es un idioma que todos hablen con fluidez y aún no existen herramientas de traducción en vivo ampliamente accesibles, lo que supone una barrera adicional.

Aun así, las videoconferencias son una de las herramientas de comunicación más frecuentes en los proyectos de código abierto. El proyecto LibreOffice, que sirve de ejemplo para esta lección, tiene reuniones online periódicas para su comunidad, por ejemplo, para desarrolladores, marketing, infraestructura, control de calidad, experiencia de usuario y diseño.

Comunicación asincrónica

La comunicación asincrónica te permite participar en la conversación en el momento y a la velocidad que te resulte más conveniente. El ejemplo más conocido es probablemente un mensaje de correo electrónico: puedes responder a un mensaje cuando lo prefieras, ya sea al minuto, al día o a la hora siguiente.

En el caso de la comunicación asincrónica, el contenido suele estar escrito, lo que además abre la posibilidad de la traducción automática. Esto te ayuda a leer y comprender mensajes escritos en un idioma diferente al tuyo, e incluso puedes traducir tu respuesta.

Además, el contenido que ya está escrito facilita mucho la retención de conocimientos y la producción de documentación. Imagina que quieres escribir un documento de soporte sobre cómo utilizar una función específica de un software. Si se lo explicas a un usuario por teléfono, será mucho más difícil convertir tu discurso en una página de documentación adecuada que si explicas el procedimiento en texto escrito.

Comunicación interna versus comunicación externa

Por último, pero no por ello menos importante, la comunicación también depende de los destinatarios previstos, es decir, si es interna o externa. La forma de escribir una nota técnica interna para los administradores del sistema es diferente a la de escribir un comunicado de prensa que se envía a cientos de periodistas. Sin embargo, tenga en cuenta que, debido a la naturaleza de un proyecto de código abierto, la comunicación que no esté explícitamente destinada a un público objetivo será visible públicamente, por ejemplo, en los archivos de listas de correo (en el caso de LibreOffice, es https://listarchives.documentfoundation.org/).

Herramientas para la comunicación

Con estos diferentes aspectos generales de la comunicación en mente, aprenderá en las siguientes secciones sobre varias herramientas de comunicación que se utilizan comúnmente en un proyecto de código abierto.

Correo electrónico, listas de correo y boletines informativos

Una de las primeras herramientas con las que te encontrarás cuando participes en un proyecto de código abierto es el correo electrónico “clásico”. Muchos proyectos utilizan listas de correo, que son básicamente listas de distribución de correo electrónico: con un mensaje de correo electrónico, puedes llegar a cientos o incluso miles de suscriptores que están interesados en temas específicos.

Las listas de correo se encuentran entre las herramientas más antiguas que se conocen en cualquier proyecto de código abierto y se utilizan para la coordinación interna del proyecto, así como para interactuar con los usuarios. Si tiene una pregunta sobre el software o desea informar un error en el programa, es muy probable que exista una lista de correo que le permita hacerlo. La comunidad de LibreOffice, por ejemplo, ofrece una variedad de listas de correo internacionales y locales para varios temas (https://www.libreoffice.org/get-help/mailing-lists/), que van desde soporte al usuario hasta discusiones con desarrolladores y coordinación de infraestructura (<<lo_mailinglists> >).

Página web de listas de correo de LibreOffice image::./images/lo_mailinglists.png[width="75%",id="lo_mailinglists"] Cada mensaje enviado se suele almacenar en un archivo de listas de correo públicas para futuras referencias. Una vez enviado, un mensaje no se puede borrar fácilmente. Una frase común dice: “Internet nunca olvida”. Por lo tanto, es recomendable tener cuidado con lo que escribes, porque probablemente no puedas retractarte. Por ejemplo, algunas personas tienen su dirección privada o número de teléfono en la firma, o envían documentos confidenciales como archivos adjuntos, lo cual debes evitar cuidadosamente.

Una desventaja de las listas de correo es que su gestión en el programa de correo no siempre es sencilla. Es posible que desee crear los llamados *filtros* basados en elementos específicos del mensaje, por ejemplo, un prefijo en la línea de asunto. Las sutilezas de la gestión de grandes cantidades de correo electrónico pueden ser una barrera para los usuarios inexpertos, especialmente si su mensaje es único. Por lo tanto, cada vez más proyectos migran a foros de discusión, sobre los que pronto aprenderá.

Una forma especial de lista de correo es un boletín informativo. Si desea mantenerse al día sobre los últimos avances del proyecto y recibir información sobre nuevos lanzamientos de software, puede suscribirse al boletín informativo y recibir un mensaje de correo electrónico cuando ocurra algo importante.

Foros de discusión

Aparte de la sobrecarga que supone gestionar listas de correo en tu cliente de correo electrónico, otra desventaja del correo electrónico es que cada vez más gente, especialmente la generación más joven, ya no se comunica tanto por correo electrónico. Por esta y otras razones, cada vez más proyectos de código abierto trasladan su comunicación a *foros de discusión*. La idea general es muy similar a la del correo electrónico: cada foro tiene varias categorías con temas específicos para debatir, los llamados *hilos*. De forma similar al correo electrónico, en un foro puedes ponerte en contacto con el proyecto de código abierto y coordinar actividades, hacer sugerencias sobre la dirección del proyecto e informar de errores como usuario final.

Todo lo que se publica en un foro suele ser visible para el público en general, al igual que en una lista de correo; pero a diferencia de lo que ocurre en este último caso, dependiendo de la configuración del foro, los contenidos también se pueden editar o eliminar más tarde. La usabilidad de los foros es, especialmente para los usuarios inexpertos, a menudo mejor que la de las listas de correo. El proyecto LibreOffice comenzó a convertir varias de sus listas de correo en foros (https://community.documentfoundation.org/) y desde entonces ha visto un aumento en la participación en la discusión.

Página web de foros de LibreOffice image::./images/lo_forums.png[width="75%",id="lo_forums"]

Mensajería instantánea y plataformas de chat

Otra forma de ponerse en contacto con una comunidad abierta es a través de mensajes instantáneos y plataformas de chat. Con el auge de herramientas populares como WhatsApp, Telegram, Signal y Matrix, casi todo el mundo ya ha instalado una de estas aplicaciones populares en sus dispositivos, lo que hace que la barrera de entrada sea mucho menor. Los mensajes instantáneos también son mucho más populares entre las generaciones más jóvenes que el correo electrónico o los foros. Por lo tanto, no es de extrañar que muchos proyectos de código abierto los adopten en la actualidad.

Los participantes de un chat escriben mensajes que se envían a todos los demás participantes, de forma similar al correo electrónico. Según la plataforma de chat, un mensaje puede enriquecerse con formato, gráficos y archivos adjuntos.

En términos estructurales, las aplicaciones de mensajería están organizadas de manera similar a un foro o correo electrónico. Hay varios *grupos* o *canales* disponibles, por lo que puedes participar en debates sobre temas que te interesen. Los mensajes generalmente se pueden editar o eliminar y, a menudo, también hay canales exclusivos de anuncios que tienen una función similar a los boletines informativos por correo electrónico.

Una desventaja de las aplicaciones de mensajería instantánea es que la gente suele instalarlas en sus teléfonos, por lo que reciben notificaciones de cada mensaje enviado. Esto puede convertirse rápidamente en un exceso de información o en una “fatiga de alertas”. Sin embargo, con una configuración adecuada, estas notificaciones se pueden mantener bajo control.

Otra desventaja es que muchas aplicaciones de mensajería son propietarias y están en manos de un solo proveedor. Esto hace que preservar la información a largo plazo sea más complicado si los datos no son de libre acceso.

Comunicaciones autónomas, federadas y centralizadas

Los proyectos de código abierto funcionan de forma abierta, basándose en estándares y herramientas abiertos. Por lo tanto, es fundamental comprender cómo se diseñan las distintas herramientas en relación con la interoperabilidad. Las opciones se pueden dividir en tres categorías principales.

Una plataforma *autónoma* funciona de manera aislada para una comunidad. Algunos ejemplos son los foros o wikis, que normalmente no están conectados con instancias de otros proyectos.

Los sistemas *descentralizados* o *federados* funcionan de forma individual para cada comunidad, pero pueden conectarse entre sí. Un ejemplo es el correo electrónico, ya que un servidor de correo electrónico local puede enviar correos a cualquier otro servidor de correo del mundo. Otros ejemplos son Nextcloud y ownCloud, que pueden "federar" archivos compartidos con otros servidores, y el servicio de mensajería Element, con el que se puede comunicar con usuarios de otros servidores. Los mismos principios se aplican a la red social Mastodon.

Tanto las plataformas independientes como las distribuidas tienen una gran ventaja: el proyecto de código abierto conserva el control total sobre todo el contenido y la funcionalidad. Todo el conocimiento almacenado en un sistema de este tipo sigue siendo propiedad de la comunidad de código abierto y no está sujeto a terceros.

Por otro lado, un sistema *centralizado* es administrado por un proveedor y no interactúa con terceros. Los ejemplos clásicos son las redes sociales como Facebook o Instagram, o incluso las aplicaciones de mensajería como WhatsApp o Telegram. Todo el contenido se almacena en los servidores del proveedor externo y está sujeto a sus términos y condiciones.

Si participa activamente en una comunidad de código abierto, es probable que se ponga en contacto con ellos mediante estas tres opciones. Los sistemas centralizados son excelentes para llegar a la gente, ya que suelen ser populares y tienen una gran base de usuarios. Sin embargo, para el trabajo real en el proyecto, lo mejor es un sistema federado o independiente bajo el control de la comunidad.

Herramientas para la colaboración

La distinción entre herramientas de comunicación y herramientas de colaboración no siempre es sencilla. A los efectos de esta lección, el objetivo principal de las herramientas de comunicación es permitir la interacción general entre diversos participantes, mientras que el objetivo principal de las herramientas de colaboración es ayudar a las personas a trabajar juntas.

Las herramientas de colaboración adecuadas permiten almacenar archivos, colaborar en tiempo real con documentos, hacer un seguimiento de las versiones de los documentos y de los cambios

entre versiones de software, y mucho más. Si bien el correo electrónico o los foros pueden servir como almacenamiento de uso general, las herramientas especializadas hacen que el conocimiento sea más accesible y la colaboración mucho más eficaz.

En otras palabras, son herramientas especializadas para tareas específicas. Si quieres contribuir a un proyecto de código abierto, te encontrarás con ellas muy pronto.

Wikis

Una de las herramientas de colaboración más antiguas y populares es la *wiki*. Wikipedia se hizo famosa porque permite a los usuarios trabajar juntos en un sitio web que está compuesto por varios documentos o "artículos". Se pueden agrupar en varias categorías, filtrar por idioma y contener formatos, hojas de cálculo e imágenes.

Las wikis se utilizan a menudo como una base de conocimiento en la que todo el mundo puede contribuir. Si quieres contribuir con contenido a un proyecto de código abierto, participar en su wiki es una de las formas más sencillas. Puedes tomar contenido existente y traducirlo a tu lengua materna, editar y actualizar artículos existentes o crear contenido nuevo. En la wiki del proyecto LibreOffice (https://wiki.documentfoundation.org), puedes encontrar material de marketing, actas de reuniones de la junta directiva, instrucciones de instalación y planificación de conferencias, todo ello en una variedad de idiomas (<<lo_wiki> >).

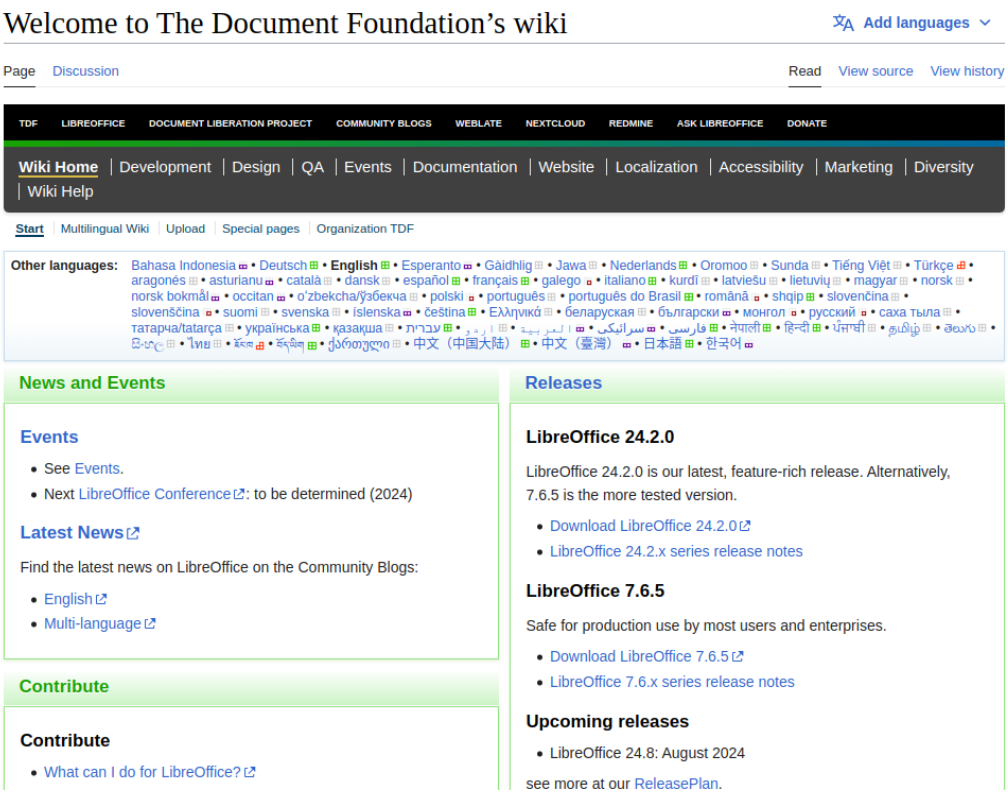

Figure 17. LibreOffice Wiki

Muchos proyectos de código abierto también alojan su documentación y el sistema de ayuda integrado de sus programas en una wiki. Además, las versiones antiguas de una página (denominadas *revisiones*) se archivan para futuras referencias.

Aunque las wikis también pueden almacenar archivos de proyecto, existen mejores herramientas para este propósito, como aprenderá en esta lección.

Rastreadores de errores y problemas

Otra herramienta que se utiliza con frecuencia en un proyecto de código abierto son los *bug trackers*, también llamados *issue trackers*. Si descubre un problema en el software o desea sugerir una nueva función, puede pensar en enviar un mensaje de correo electrónico al respecto. Sin embargo, con una herramienta dedicada como un *bug tracker*, puede proporcionar toda la información y los pasos necesarios para reproducir un problema de forma estructurada, lo que facilita a los desarrolladores la reproducción del problema. Un informe estructurado de este tipo se denomina *bug report*.

Además, la información no se pierde y el proyecto no se olvida del problema; puede asignarlo a la persona adecuada y ver cuántos errores se están procesando o solucionando.

El proyecto LibreOffice proporciona un rastreador de errores que está abierto a la contribución de todos (https://bugs.documentfoundation.org).

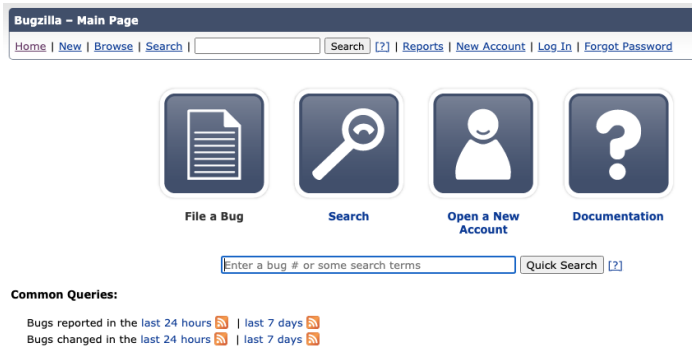

Figure 18. Rastreador de errores de LibreOffice

Helpdesk y sistemas de tickets

Una herramienta muy similar es un *helpdesk* o *sistema de tickets*. Su objetivo no es tanto informar sobre problemas de software, sino ayudar a los usuarios con todo tipo de problemas y solicitudes, como por ejemplo el sitio web del proyecto.

El Helpdesk clásico es una línea directa de atención al cliente. Un cliente llama y notifica un problema, que se convierte en un ticket. El flujo de trabajo de un sistema de Helpdesk suele girar en torno a prioridades, niveles de escalamiento y tiempo de respuesta.

No todas las comunidades de código abierto ofrecen un sistema de este tipo, pero muchas empresas comerciales sí lo hacen. Por ejemplo, en el caso del proyecto LibreOffice, existe un sistema de tickets para que la infraestructura informe sobre problemas con los servidores y los servicios web.

Sistema de gestión de contenidos (CMS)

Otra herramienta importante para la colaboración es un sistema de gestión de contenido (CMS). Como su nombre indica, ayuda a gestionar el contenido, sobre todo en el caso de los sitios web. Si quieres contribuir al contenido o al diseño del sitio web de un proyecto, familiarizarte con su CMS es el camino a seguir.

Los sistemas CMS, similares a las wikis, ayudan a estructurar el contenido en categorías e idiomas. Suelen ofrecer un editor WYSIWYG (“what you see is what you get` que Inglés significa `lo que ves es lo que obtienes”) e incrustan todo en una plantilla adecuada: los títulos, encabezados, diseño de página y entradas de menú que se realizan automáticamente, para que puedas concentrarte por completo en el contenido. Además, en caso de rediseñar una página, el contenido no desaparecerá, sino que se adaptará a la nueva plantilla.

Sistema de Gestión Documental (DMS)

Un sistema de gestión de documentos no debe confundirse con un sistema de gestión de contenidos. Mientras que un CMS está diseñado para mostrar contenidos en una plantilla predefinida, como para presentar un sitio web, un DMS se utiliza para la gestión de documentos como contratos, facturas, recibos, mensajes de correo electrónico y todo tipo de correspondencia.

Otra diferencia es que un CMS se utiliza a menudo para presentaciones públicas, mientras que un DMS se utiliza principalmente para gestionar documentos internos que no están destinados al público en general.

Como colaborador de un proyecto de código abierto, es menos probable que se encuentre con el sistema de gestión de documentos, ya que a menudo está reservado para roles específicos, como contabilidad o legal.

Gestión del código fuente (SCM)

Si eres desarrollador de un proyecto de código abierto, una de las herramientas clave con las que te encontrarás pronto es la plataforma de gestión de código fuente. Los desarrolladores de software utilizan plataformas de gestión de código fuente que, al igual que una wiki para autores de documentación, se utilizan para trabajar en conjunto con el código.

Los SCM históricos incluyen CVS y Subversion, pero hoy en día Git es el más utilizado, incluso por la comunidad de LibreOffice (https://git.libreoffice.org/). Estas herramientas están disponibles en la línea de comandos, pero también hay interfaces gráficas disponibles para facilitar la interacción, especialmente para principiantes.

Las plataformas de gestión de código fuente rastrean diferentes versiones de cada archivo, manejan cambios y ediciones del código, registran quién realizó qué cambio e idealmente brindan un historial completo del desarrollo del software.

Los desarrolladores pueden “extraer” un estado específico del software, trabajar en él localmente (por ejemplo, corrigiendo un error o implementando una nueva característica) y luego solicitar que este cambio se agregue a la línea de desarrollo principal del software mediante una llamada *solicitud de fusión*, también conocida como *solicitud de extracción*. Al aceptar la solicitud de fusión o de extracción, se “fusionan” los cambios realizados por un autor con el código principal.

Existen plataformas centralizadas (GitHub y GitLab) que integran SCM con wikis, seguimiento de problemas y otras herramientas colaborativas.

Las plataformas de gestión de código fuente no se limitan estrictamente al código de programa. Por ejemplo, ¡esta lección se desarrolló de manera colaborativa en un repositorio Git!

Documentación

La clave del éxito de cualquier proyecto de código abierto es una documentación adecuada, idealmente en varios idiomas. El proyecto LibreOffice ofrece libros, guías y tarjetas de referencia actualizados (https://documentation.libreoffice.org) para su software, así como páginas de ayuda individuales sobre funciones específicas (https://help.libreoffice.org).

En general, existen diferentes tipos de documentación, que discutiremos en las siguientes secciones.

Documentación del usuario

La documentación más conocida es la destinada a los usuarios finales, que explica cómo utilizar el software. Si no conoce alguna característica o función del programa, la documentación (que puede ir desde páginas de ayuda individuales hasta un libro completo) es su primer punto de referencia.

El sitio web de documentación, donde se explica el uso del software, suele ser uno de los sitios web más visitados junto al sitio web del producto, que presenta una descripción general del software y su comunidad.

Documentación del administrador

Para su uso en entornos más grandes, como las implementaciones en una empresa, la documentación del administrador proporciona toda la información relevante para implementar el software a mayor escala. Esto incluye la conexión a bases de datos de usuarios y almacenamiento de archivos, la gestión centralizada de la configuración y el manejo de actualizaciones.

Documentación para desarrolladores y arquitectura

Otra categoría de documentación está dirigida a los desarrolladores y se denomina documentación para desarrolladores o documentación de arquitectura. Si eres desarrollador y quieres contribuir al código de un programa, esta documentación te informa sobre la arquitectura del software, los estándares de codificación, las herramientas y flujos de trabajo que se utilizan para trabajar en el software.

El proyecto LibreOffice ha publicado una guía para desarrolladores en su wiki para ayudar a los desarrolladores interesados a unirse a la comunidad (https://wiki.documentfoundation.org/Documentation/DevGuide).

Ejercicios guiados

1. ¿Por qué los proyectos de código abierto en particular tienen que ocuparse de herramientas adecuadas para la comunicación y la colaboración?

2. Mencione un ejemplo de comunicación sincrónica y asincrónica.

3. ¿Cuál es una desventaja de las aplicaciones de mensajería y cómo se puede evitar?

4. Nombra dos funciones de una wiki.

5. ¿Cuál es la diferencia entre un sistema de seguimiento de errores y un sistema de soporte técnico?

6. ¿Cuál es la diferencia entre un sistema de gestión de contenidos y un sistema de gestión documental?

7. ¿Cuál es la ventaja de los sistemas independientes o federados sobre los sistemas centralizados?

Ejercicios exploratorios

1. ¿Cuál es una de las principales diferencias entre un club deportivo local y un proyecto internacional de código abierto?

2. ¿Por qué puede ser particularmente gratificante contribuir a un proyecto de código abierto?

3. ¿Qué software de seguimiento de errores utiliza el proyecto Ubuntu?

4. ¿Cuál es el nombre del sitio web de las listas de correo del kernel de Linux?

Resumen

En esta lección, aprendiste acerca de una variedad de herramientas que se utilizan para comunicarse y colaborar en un proyecto de código abierto. Escuchaste la diferencia entre comunicación sincrónica, asincrónica y entre herramientas descentralizadas, centralizadas e independientes. También aprendiste por qué herramientas especiales pueden ser útiles para tareas específicas para que contribuir a un proyecto de código abierto sea divertido y gratificante.

Respuestas a ejercicios guiados

1. ¿Por qué los proyectos de código abierto en particular tienen que ocuparse de herramientas adecuadas para la comunicación y la colaboración?

   Por un lado, trabajar juntos en un grupo distribuido mundialmente tiene su conjunto de desafíos, que las herramientas adecuadas pueden ayudar a abordar. Por otro lado, los voluntarios pueden no quedarse indefinidamente, por lo que retener y compartir el conocimiento es otro aspecto importante en el uso de herramientas de comunicación y colaboración. Facilitar las contribuciones ayuda a la sostenibilidad de un proyecto.

2. Mencione un ejemplo de comunicación sincrónica y asincrónica.

   La comunicación sincrónica puede ser una conversación directa, una llamada telefónica o una videoconferencia. Algunos ejemplos de comunicación asincrónica son el correo electrónico, los mensajes SMS, las cartas postales y los faxes.

3. ¿Cuál es una desventaja de las aplicaciones de mensajería y cómo se puede evitar?

   Cuando se instalan en el teléfono, es posible que recibas muchas notificaciones, una por cada mensaje nuevo. Una configuración adecuada puede ayudar a evitar esto. Otra desventaja es que muchas aplicaciones de mensajería son administradas por proveedores propietarios.

4. Nombra dos funciones de una wiki.

   Edición colaborativa y traducción de artículos.

5. ¿Cuál es la diferencia entre un sistema de seguimiento de errores y un sistema de soporte técnico?

   Un sistema de seguimiento de errores es una herramienta de software especializada para informar errores o solicitar funciones en el software. Un sistema de soporte técnico se centra en dar soporte a consultas y gestionar todo tipo de problemas y solicitudes, ejemplo, en el sitio web.

6. ¿Cuál es la diferencia entre un sistema de gestión de contenidos y un sistema de gestión de documentos?

   Un CMS se utiliza para presentar contenido de una forma o plantilla específica, generalmente de forma pública. Un DMS se utiliza para almacenar correspondencia existente, generalmente de forma interna.

7. ¿Cuál es la ventaja de los sistemas independientes o federados con respecto a los sistemas

centralizados?

Un sistema centralizado está bajo el control de un proveedor externo. Todo el contenido se almacena en los servidores del proveedor externo y está sujeto a sus términos y condiciones.

Respuestas a ejercicios exploratorios

1. ¿Cuál es una de las principales diferencias entre un club deportivo local y un proyecto internacional de código abierto?

   Los proyectos de código abierto no están ligados a un idioma o ubicación específicos. Los colaboradores pueden vivir en diferentes países y continentes, tener diferentes lenguas maternas e incluso vivir en diferentes zonas horarias. La mayor parte de la actividad que se lleva a cabo en un proyecto de código abierto ocurre virtualmente, no en persona.

2. ¿Por qué contribuir a un proyecto de código abierto puede ser particularmente gratificante?

   Muchos proyectos de código abierto cuentan con un grupo diverso de personas. Al colaborar con ellos, puedes aprender de ellos, descubrir cosas nuevas y ampliar tu alcance.

3. ¿Qué software de seguimiento de errores utiliza el proyecto Ubuntu?

   Launchpad

4. ¿Cuál es el nombre del sitio web de las listas de correo del kernel de Linux?

   `https://lkml.org/`

Pie de imprenta

PDF generado: 2024-10-02

LPI Learning Materials son una iniciativa del Linux Professional Institute (https://lpi.org). Los materiales y sus traducciones pueden encontrarse en https://learning.lpi.org.

Para preguntas y comentarios sobre esta edición, así como sobre todo el proyecto, escriba un correo electrónico a: learning@lpi.org.