Linux
Professional
Institute

Open Source Essentials

Version 1.0
English

050

Table of Contents

Topic 051: Software Fundamentals

Linux Professional Institute

051.1 Software Components

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 051.1

Weight

2

Key knowledge areas

- Understanding the concept of source code and code execution
- Understanding the concept of compilers and interpreters
- Understanding the concept of software libraries

Partial list of the used files, terms and utilities

- Source code
- Executable programs
- Byte code
- Machine code
- Compiler
- Linker
- Interpreter
- Runtime virtual machine
- Algorithm
- Software libraries
- Static and dynamic linking

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	051 Software Fundamentals
Objective:	051.1 Software Components
Lesson:	1 of 1

Introduction

Free and *open source software* — often abbreviated as FOSS — has become an integral part of our everyday lives, mostly without us being aware of it. FOSS, for example, can be found behind all of our activities on the internet in some form: on the computer where we view web pages in the browser, or on servers that store these web pages and deliver them as soon as we call them up.

What is Software?

Before we look at all the specifics of free and open source software, however, we need to clarify what *software* actually is. Let's start with a very general description: Software is the non-physical, immaterial part of computers in any form. Software ensures that the physical parts (the *hardware*) of the computer interact and that the computer can accept commands and execute tasks.

A smartphone, a laptop, or a server in the data center is just a machine made of metal and plastic when it is switched off. As soon as it is turned on, software starts up, in the form of coded command sequences that control the individual components of this machine, that allow the user to interact with the machine, and that perform very specific tasks by invoking individual applications.

It is the work of *software developers* to analyze the tasks that the computer is supposed to perform and to specify them in a way that allows the computer to implement them. The tools used by the developers are as numerous and diverse as the tasks performed by the software.

Some software tasks are very closely connected to the hardware and the architecture of the computer, e.g., the addressing and management of memory or the handling of different processes. *System programmers* therefore work close to the hardware.

Application developers, on the other hand, focus more on the user and program applications that enable users to perform their tasks both efficiently and intuitively. An example of a complex application is a word processing program that provides all the functions for text formatting in menus or buttons and also displays the text as it might ultimately be printed.

In general, an *algorithm* is a way of solving a problem. For instance, to calculate an average, the normal algorithm is to add a collection of values and divide the sum by the total number of values. Although algorithms are traditionally designed and carried out by programmers, algorithms are also generated nowadays by artificial intelligence.

The concepts in this chapter can help you understand the strengths and risks of FOSS, make informed choices, and even decide whether you want to be a software developer.

Programming Languages

Programming languages are highly structured, artificial languages that tell a computer what to do. Programs are usually written in text, but some languages are written in graphical form. Software developers write instructions (called *code*) to the computer in this artificial language. However, the computer hardware does not directly execute this code. Hardware can directly execute only a series of bit patterns stored in memory, called *machine code* or *machine language*. All programming languages are either converted into machine code by a *compiler* or interpreted by another machine code program called an *interpreter* to make the hardware execute these instructions.

Some of the most widely used programming languages currently include Python, JavaScript, C, C++, Java, C#, Swift, and PHP. Each of these programming languages has its own unique strengths and weaknesses, and the choice of language depends on the project and the needs of the developer. For example, Java is a popular choice for developing large-scale enterprise applications, while Python is often used for scientific computing and data analysis.

Developers have shown impressive creativity in designing programming languages. Originally, they were *low-level languages* that resembled the instructions in the computer. Languages have become more and more *high-level*, meaning that they try to represent powerful combinations of instructions is brief terms. Some languages reflect the way people naturally think, while preserving the rigor necessary to run correctly.

About 400 programming languages are currently recognized, although many are used only in very niche applications or legacy environments. Each was developed in order to solve in certain tasks.

Characteristics, Syntax, and Structure of Programming Languages

The choice of programming language can have a significant impact on performance, scalability, and ease of development in a software project. These sections lay out important elements of languages.

Characteristics of Programming Languages

Some of the common characteristics and qualities of programming languages include:

Concurrency

Concurrency denotes handling multiple tasks simultaneously, either by running on different hardware processors or by alternating the tasks' use of a single processor. The degree of concurrency supported by a programming language can greatly affect its performance and scalability, especially for applications that require real-time processing or large amounts of data. Each separate piece of work might be called a *process*, a *task*, or a *thread*.

Memory management

Memory management is allocation and freeing of memory in a program. Depending on the language or runtime environment, memory management can be done manually by the programmer or handled automatically. Proper memory management is crucial for ensuring that a program uses memory effectively, and that it does not run out of memory or cause other problems. If a program fails to free unused memory, the program causes a *memory leak* that gradually increases the use of memory until the program crashes or negative performance effects are noticeable.

Shared memory

Shared memory is a type of interprocess communication mechanism that enables multiple processes to read and manipulate a common region of memory. Shared memory is common in hardware such as disk drives, and can also be an efficient way to share data between processes. But the mechanism requires careful synchronization and management to prevent data corruption. An error known as a *race condition* occurs if one process makes an unanticipated change to data while another process is using it.

Message passing

Message passing is a communication mechanism between processes that enables them to exchange data and coordinate their activities. This is commonly used in concurrent programming to achieve interprocess communication, and can be implemented through various mechanisms such as sockets, pipes, or message queues.

Garbage collection

Garbage collection is an automatic memory management technique used by some programming languages to reclaim memory that is no longer being used while a process is running. This can help prevent memory leaks and make it easier for developers to write correct and efficient code, but it can also introduce performance overhead and make control over the precise behavior of the program more difficult.

Data types

Data types determine what type of information can be represented in the program. The data types can be predefined in the language or user-defined, and can include integers, floating-point numbers (i.e., approximations of real numbers), strings, arrays, and others.

Input and output (I/O)

Input and output are mechanisms for reading and writing data to and from a program. Input can come from a variety of sources, such as user clicks and keyboard input, a file, or a network connection, while output can be sent to a variety of destinations, such as a display, a file, or a network connection. I/O allows programs to interact with the outside world and exchange information with other systems.

Error handling

Error handling detects and responds to errors that occur during the execution of a program. This includes errors such as division by zero and a requested file that is not found. Error handling allows programs to continue running even when errors occur, improving their reliability and robustness.

The concepts just listed are fundamental to understanding how programming languages work and how to write efficient and maintainable code.

Syntax of Programming Languages

The *syntax* of a programming language refers to rules for writing program statements and expressions. It is important for the syntax to be *well-defined* and *consistent*, so that the programmer can effectively write and understand their code. The following are the building blocks of most programming languages:

Procedures and functions

Procedures and functions are used to define reusable blocks of code that can be called multiple times.

Variables

Variables represent pieces of memory, and store data that can be manipulated and passed between procedures and functions.

Operators

Operators are keywords or symbols (such as `+` and `-`) that assign values to variables and perform arithmetic operations.

Control structure

Generally, program code is executed in the order that it is written, but *conditional statements* change the flow of execution. Which code is executed next is based on various conditions, such as the contents of memory, the state of the keyboard, packets arriving from the network, and so on. The *loop statement*, a special form of conditional statement, is useful for performing the same operations on a series of data sets. An *exception*, which invokes special code when an error occurs, is another control structure.

The syntax and behavior of these constructs can vary between programming languages, and the choice of language can have a big impact on the readability and maintainability of the code.

Libraries

A good programming language should make it easy to develop programs and easy to reuse existing code. Many programming languages have a mechanism to organize procedures and functions into parts that can be reused in other programs.

A *library* is a collection of procedures and functions in support of a particular feature or goal, combined into a single file. The availability of many easy-to-use libraries is another very important requirement of a good programming language. For example, Python is widely recognized as a good language for developing AI-related programs because it has a number of libraries suitable for AI processing.

With the increasing size and complexity of programs, libraries as ready-made building blocks are becoming more and more important. This is especially true in the open source world, where people are comfortable taking code that others have created and reusing it. As a result, an ecosystem of libraries has developed for each programming language, and package managers such as `composer` for PHP, `pip` for Python, and `gems` for Ruby make it easy to install libraries.

Libraries are also important in compiled languages. Combining multiple binary files and pre-compiled libraries to obtain a single executable file is called *linking*, and the tool that performs this operation is called a *linker*. There are two types of linking: *static linking*, in which only the necessary library code is included in the final application's executable file, and *dynamic linking*, in which a library installed in the system is shared by all applications that use that library. Currently, dynamic linking is the preferred approach and is characterized by smaller application executables and less memory usage at runtime.

Note that since the same libraries can be used by multiple programs, differences between versions of a library can be even more of an issue than with applications. Let's digress for a moment and remember how to look at version numbers. *Semantic versioning* is commonly used, which indicates versions by three numbers separated by dots. A typical version might be `2.39.16`, which indicates a major version of 2 (a number that is likely to change only once every few years), a minor version of 39 within the major version (which may update every few months to contain important feature changes), and a fast-moving revision of 16 (which can change because of a single bug fix). Later versions and revisions have higher numbers.

A Very Simple Example

Let's look at a *very* simple example of a computer program in the Python language to get a rough idea of a few of the elements mentioned.

Stated in natural language, the program is supposed to do the following: “Ask the user to enter a number and check whether this number is even or odd. Finally, output the result.”

And here is the code we can save in the file `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.")
```

Even in these few lines of code, we can find many of the characteristics and syntax elements mentioned above:

1. In line 1 we set the *variable* `num` and assign it a *value* with the `=` *operator*.
2. The assigned value corresponds to the *input* of the user (via the `input()` *function*). In addition, the `int()` function ensures that this input is converted to the integer *data type*, if possible. The expression that is passed to a function within parentheses is called a *parameter* or *argument*.

3. If the user enters a string of letters, Python would print an error as part of its *error handling*. If a decimal number is entered, the function `int()` converts it to the base number, e.g. `5.73` to `5`.
4. In the following lines, the *control structure* stating a *condition*, with the keywords `if` and `else`, controls what happens in each of the two possible cases (the number is either even or odd).
5. First, the modulo operator `%` tests whether (`if`) dividing the entered number by 2 yields the value 0 (i.e., no remainder) — in this case, the number is even. The doubled `==` is the “is equal to” comparison operator, which is different from the assignment operator `=` in line 1.
6. In the other case (`else`), i.e. when the division by 2 produces a result unequal 0, the entered number must be odd.
7. In both cases, the function `print()` returns the result as *output* in text form.

And this is what it looks like when we run the program on the command line:

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

When you consider how much language logic is already involved in this small example, you gain an idea of what complex software distributed over thousands of files is capable of; for example, *operating systems* such as Microsoft Windows, macOS, or Linux, which make all the hardware of a computer available and at the same time ensure that the users can install all other desired applications and use them for work or fun.

Machine Code, Assembly Language and Assemblers

As mentioned earlier, hardware can directly execute only a series of bit patterns called machine code. The CPU reads a bit pattern from memory in units of a word (8 to 64 bits) and executes the correlating instruction. Individual instructions are quite simple, for example, “copy the contents of memory location A to memory location B,” “multiply the contents of memory location C by the contents of memory location D,” or “read the data that has arrived at the device at address X.” In the era of 8-bit CPUs, some people could memorize all the bit patterns used in machine code and write programs directly. Nowadays, the number of instruction patterns has increased by an order of magnitude, and trying to remember all of these patterns is impractical.

Machine code is a sequence of bit patterns, or 0s and 1s, which is not at all intuitive to humans. To make programming more intuitive, *assembly language* was created, in which the instructions were given names and could be specified by strings. In assembly language, instructions that correspond one-to-one to the machine code are written one at a time. Instructions might look like:

```
move    [B], [A]           copy the contents of memory A to memory B
multi   R1, [C], [D]    multiply the contents of memory C by the contents of memory D
input   R1, [X]         read the data that has arrived at the device at address X
```

One instruction in assembly language corresponds to one instruction in machine code, which corresponds to the exact instruction that the hardware can understand and perform. Advantages of assembly language over machine language include:

Improved readability and maintainability

Assembly language is much easier to read and write than machine code. This makes it easier for programmers to understand, debug, and maintain their code.

Address computation automation

Machine code programming can also use the concept of variables and functions, but everything must be expressed in terms of *memory addresses*. Assembly language also assigns names to memory addresses, which makes it easier to express the logic of a program.

Because assembly language has access to all of the hardware's functionality, it is commonly used in the following situations:

Architecture dependent part of the operating system

Using dedicated instructions that are specific to one CPU architecture, for access to initialization and security features, can be done only in assembly language.

Developing low-level system components

Assembly language is used to develop system components that need to interact directly with the computer's hardware, such as device drivers, firmware, and the basic input/output system (BIOS). In particular, high-speed devices that require pushing hardware performance to its limits often need drivers and firmware programmed in assembly language.

Programming microcontrollers

Assembly language is also used to program microcontrollers, which are small, low-powered computers used in a wide range of embedded systems from toys to industrial control. Some microcontrollers have memory capacities of just several hundred bytes, and are commonly programmed in assembly language.

Assembly language code is converted to machine code before being executed by an application called an *assembler*. The assembler is the oldest programming tool and has brought a number of advantages that were unthinkable in machine code programming. To confuse matters, sometimes people refer to assembly language as assembler.

Machine code and assembly language differ from one hardware processor to another. They are called “low-level languages” because they operate directly on hardware. However, the concepts of computation and input/output are the same across all processors. If the common concepts can be expressed in a way that is easier for humans to understand, programming efficiency can be dramatically improved. This is where “high-level languages” come in.

Compiled Languages

Compiled languages are programming languages that are translated either into machine code or into an intermediate format called *bytecode*. Bytecode is executed on the target computer by a *runtime virtual machine*. The virtual machine translates the bytecode into the proper machine code for each computer. Bytecode allows programs to be platform-agnostic and to run on any system with a compatible virtual machine.

The translation of the source code written in a high-level programming language into machine code or bytecode is done by a *compiler*. Examples of compiled languages that produce machine code directly include C and C++. Languages that produce bytecode include Java and C#.

The choice between machine code and bytecode depends on the requirements of the project, such as performance, platform agnosticism, and ease of development.

Interpreted Languages

Interpreted languages are programming languages that are executed by an *interpreter*, rather than being compiled into machine code. The interpreter reads the source code and executes the statements contained in it. An interpreter is able to directly process the source code, without transforming it into another file format. Thus, unlike a compiler, which translates the entire program into machine code before executing it, an interpreter reads each line of code and executes it immediately, allowing the programmer to see the results of each line as they are executed.

Interpreted languages are commonly used for *scripting*, which refers to short programs that automate tasks, for command-line interfaces, and for batch and job control. Scripts written in interpreted languages can be easily modified and executed without the need for recompilation, making them well suited for tasks that require rapid prototyping or flexible and rapid iteration. With this convenience come some potential drawbacks. For instance, an interpreted program runs more slowly than its equivalent compiled program.

Examples of interpreted languages include Python, Ruby, and JavaScript. Python is widely used in scientific computing, data analysis, and machine learning, while Ruby is often used in web development and for creating automation scripts. JavaScript is a client-side scripting language embedded in web browsers to create dynamic and interactive web pages.

Data-Oriented Languages

Data-oriented languages are programming languages that are optimized for processing and manipulating large amounts of data. They are designed to efficiently handle large sets of structured or unstructured data and provide a set of tools for working with databases, data structures, and algorithms for data processing and analysis.

Data-oriented languages are used in a variety of applications, including data science, big data analytics, machine learning, and database programming. They are well suited for tasks that involve processing and analyzing large amounts of data, such as data cleaning and transformation, data visualization, and statistical modeling.

Examples of data-oriented languages include SQL (short for Structured Query Language), R, and MATLAB. SQL is a standard language used for managing relational databases and is widely used in business and industry. R is a programming language and environment for statistical computing and graphics and is widely used in data science and machine learning. MATLAB is a numerical computing environment and programming language used in a wide range of applications, including signal processing, image processing, and computational finance.

Programming Paradigms

Besides the specifics of programming languages, *programming paradigms* determine the particular solution approach. We can think of a paradigm as a basic strategy with which we approach a task, depending on the specific requirements and conditions.

A comparable example is the construction of a house: Whether masons erect the walls brick by brick or ready-cast concrete components are assembled on site is a fundamental decision that depends on requirements and circumstances. What features do you want the house to have? Where is it located? Is it connected to other houses?

In a similar way, paradigms set the direction of programming: whether and in what way, for example, a software project is broken into smaller, separate parts. Each programming language is suited best to some particular paradigm. Therefore, the choice of paradigm is closely related to the choice of programming language.

The following paradigms are common in programming:

Object-oriented programming (OOP)

OOP is based on the concept of *objects*, which are instances of *classes* that encapsulate data and behavior. For instance, a language might offer a rectangle as a class to help the programmer display a box on the screen.

OOP focuses on the object-level manipulation of data. OOP makes it easier to write code that is maintainable, reusable, and extensible, and is widely used in desktop software, video games, and web applications. Examples of object-oriented programming languages include Java, C#, and Python.

Procedural programming

Procedural programming performs tasks through *procedures*, or blocks of code that can be executed in a specific order. This makes it easy to write structured code that is easy to follow, but can lead to code that is less flexible and harder to maintain as the size and complexity of the project grows. Examples of procedural programming languages include C, Pascal, and Fortran.

Other approaches to software development are in use today, for which some languages are better suited than others. Additionally, drag-and-drop interfaces allow non-programmers to write programs, and many online services have recently started to generate code through artificial intelligence when given plain-language instructions.

In conclusion, each programming paradigm has its own strengths and weaknesses, and the choice of paradigm often depends on the needs of the project, the experience and preferences of the developer, and the constraints of the platform and development environment. Understanding the different types of paradigms can help you choose the right paradigm for your needs, and can also help you write better and more efficient code.

Guided Exercises

1. What is the purpose of functions?

2. What is the advantage of bytecode over a machine code file?

3. What is the advantage of a machine code file over bytecode?

Explorational Exercises

1. What are some disadvantages of dividing a program into a large number of processes or tasks?

2. You have found several open source packages, offered in different versions, that provide features you need for your program. What are some criteria for choosing a package?

3. In addition to OOP and procedural development paradigms, what other software development approaches exist and what is a programming language which best supports each approach?

Summary

In this lesson you have learned what software is and how it is developed with the help of programming languages. The numerous programming languages differ not only in their syntax, but also, for example, in their management of hardware resources or handling of data structures.

Programming languages also differ in how the source code, readable by humans, is converted by an interpreter or compiler into the final machine code to be processed by the computer.

Programming paradigms determine the strategy of software projects and thus also the choice of suitable programming languages, depending on the requirements and the size of the respective project.

Answers to Guided Exercises

1. What is the purpose of functions?

   Functions encapsulate certain common activities, such as outputting a string. By making a function, you can allow your program and other programs to perform the function conveniently and repeatedly without having to write their own code for it.

2. What is the advantage of bytecode over a machine code file?

   The bytecode file can run on many different computers, where a virtual machine turns the code into machine code. For instance, JavaScript runs in many browsers on many types of computers.

3. What is the advantage of a machine code file over bytecode?

   Machine code runs as fast as possible. Bytecode runs more slowly because the virtual machine must turn it into machine code while running the bytecode.

Answers to Explorational Exercises

1. What are some disadvantages of dividing a program into a large number of processes or tasks?

   When a program is divided into processes, they must communicate with each other. If they work on a lot of data in common, the processes could expend a lot of overhead in data exchange and in protecting data from multiple, simultaneous changes (race conditions). Processes also incur overhead when they start up and terminate. The more processes you have, the more complex the program and its interactions become, so errors can be harder to find.

   The functional paradigm tends to make it easier to divide programs into many processes, because of immutability. Immutable data does not suffer from race conditions.

2. You have found several open source packages, offered in different versions, that provide features you need for your program. What are some criteria for choosing a package?

   Check bug reports and security advisories for the packages, because some are very buggy and even unsafe. Sometimes the latest version is not the best, because a security flaw might have been introduced into it.

   Peruse the forum where developers talk about the package, to see that it is actively maintained. Your program will probably be in use for a long time, and you want the package to be available and robust over time as well.

   Try different packages to check their performance, as well as their correctness.

   Most packages depend on functions found in other packages (*dependencies*), so a weakness in one of the dependencies can affect your program.

3. In addition to OOP and procedural development paradigms, name some other software development approaches and a programming language that best supports each approach.

   In addition to OOP and procedural development paradigms, other types include the following.

   Functional programming emphasizes the use of functions and mathematical concepts, such as lambdas and closures, to write code that is based on the evaluation of expressions rather than the execution of statements. Functional programming treats functions as first-class citizens — so that they can be manipulated by the program — and emphasizes *immutability*, or working with variables that cannot change after being initially set. This makes it easier to reason about and test code, as well as to write concurrent and parallel applications. Examples of functional programming languages include Erlang, Haskell, Lisp, and Scheme.

Imperative languages focus on the statements required to control the flow of the program’s transitions from and to various states.

Declarative languages describe what actions to take and the logic behind the instructions. The order in which the instructions are carried out isn’t specified. These instructions, function calls, and other statements can be reordered and optimized by compilers so long as they preserve the underlying logic.

Natural programming is a programming paradigm that uses natural language or other human-friendly representations to describe the desired behavior of a program. The idea is to make programming accessible to people who may not have formal training in computer science. Examples of natural programming languages include Scratch and Alice.

051.2 Software Architecture

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 051.2

Weight

2

Key knowledge areas

- Understanding the concepts of client and server computing
- Understanding the concepts of thin and fat clients
- Understanding the concepts of monoliths and microservices and their main differences
- Understanding the concepts of Application Programming Interfaces (APIs)
- Understanding the concept of software components and their integration or separation (services, modules, APIs)

Partial list of the used files, terms and utilities

- Clients and servers
- Thin clients and fat clients
- Web applications
- Single-page applications
- Monolithic architectures
- Microservice architectures
- Application Programming Interfaces (APIs)
- RESTful APIs

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	051 Software Fundamentals
Objective:	051.2 Software Architecture
Lesson:	1 of 1

Introduction

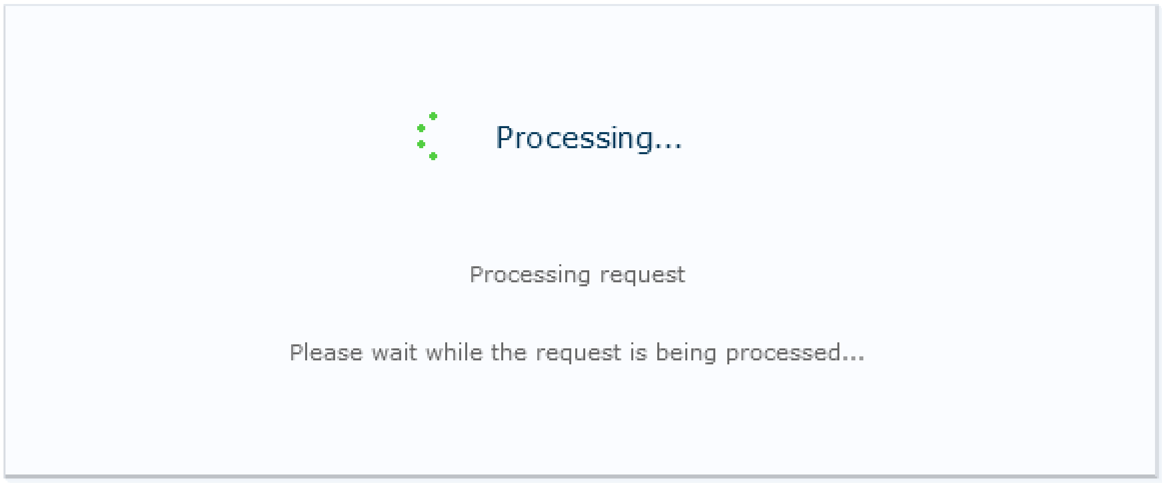
The internet is ubiquitous in our modern world, as are mobile and web-based applications. These tools are used seamlessly by a large portion of the world’s population and power everything from instant messaging to more complex activities like purchasing mining equipment for large companies.

Behind all these seemingly simple interfaces and online services is an *architecture* — an arrangement of cooperating pieces of software — that we often take for granted. You need to know a bit about this architecture to understand how all the pieces of the internet fit together and how software can deliver real value to its users.

In this lesson, we will look at some of the software architectures behind web applications, which are server-based software systems, and how they are used within systems that almost everyone is familiar with.

Servers and Clients

When you use online systems, it is highly likely that at some point you have encountered a message like the one in Sample screen shown while a response is underway.


Processing...

Processing request

Please wait while the request is being processed...

Figure 1. Sample screen shown while a response is underway

Let's step back a little bit and look at the context in which you're seeing this message. Say you are trying to gain access to your banking account through your bank website. When you try to get onto the banking website on your laptop, you use a type of software (i.e., application) called a web browser, such as Google Chrome or Firefox. In this case, the browser on your computer sends a request to another computer hosting the website. This particular kind of computer is called a *server*. It is specially designed to run 24/7, always serving new requests coming in from all parts of the world.

So a server is a computer, just like the one you use to work with, play video games, and do programming assignments. However, there is one main difference: A server normally uses all its resources for the software application running on it. In this example, the software is a web application, a computer program that runs on the server.

In Sample screen shown while a response is underway, the server has received a request from a browser, and the application running on the server is processing the resulting operation. This operating could be querying a database to fetch a user's details in the bank, or communicating with another server to verify a special discount on the user's next loan.

In this example, we call the browser running on your machine the *client application*, or simply put, the *client*. The client interacts with the remote server.

The network communication between client and server might take place inside an enterprise network, or across the worldwide network we call the internet. A common characteristic of client-server interaction is that a server can establish multiple relationships with multiple clients. Think about the previous example: the website of a bank hosted on a server can attend thousands of requests per minute from multiple locations, each with a user trying to access their personal bank account.

Not all scenarios are structured like a browser interacting with a server that does almost all the processing. In some cases, the client can be the primary instance for processing; this concept is called a *fat client* (or *thick client*), where the client stores and processes the majority of the tasks instead of relying on the server's resources. In our banking example, in contrast, the browser is a *thin client* that relies on the server to compute and return information through the network.

An example of a fat client is a video game desktop application, where the bulk of the data storage and processing is done locally, using the computer's GPU, RAM, CPU, and disk space to process the information. Such an application rarely relies on an external server, especially if the game is being played offline.

Both approaches have pros and cons: For a thick client, network instability is less of an issue compared with a thin client that relies on a remote server, but software updates can be harder to apply and the fat client requires more computer resources. For a thin client, the lower costs could be a great advantage. For either kind of client, providing personal data to a third-party application can be an issue.

Web Applications

A *web application* is software that runs on a server, processes user interactions, and is contacted by either fat or thin clients through a computer network. Not all websites are considered web applications: Simple static web pages without interactivity are not considered web applications because the server doesn't run an application to process actions requested by the client.

A lot of web applications can be divided into two groups: the *single page application* (SPA) and the *multi-page application* (MPA). An SPA has only one web page, where all data exchange and loading occur without the need for redirecting the user to another web page inside the application. An MPA, in contrast with the SPA, has multiple web pages. A data change might either refresh the same web page that originated the action or redirect the user to another web page.

Consider the previous example, where a user wants to check their most recent account transactions on the internet banking website. Imagine that a transaction happens after the web page is loaded. If the bank web application is an SPA, the new transaction will be displayed automatically on the same web page, without redirecting the user to a new page. If the user checks their loans, the new information is also displayed on the same page, avoiding the need to redirect the user to a new web page. Altering the page without redirecting the user makes the navigation smooth.

For an MPA, when the user requests the loan web page, the server must redirect the user to a new location, which means another web page.

A famous example of an SPA web application is Google Mail (Gmail). It doesn’t redirect the user to completely new page when, for example, the user wants to display the the spam folder; the application merely refreshes the specific part of the display that shows all spam messages, and stays on the same web page.

On the other hand, a famous MPA application is Amazon.com, the ecommerce giant, where each item is located on a distinct web page.

One advantage of an MPA over an SPA is that web analytics are much easier to gather and measure. This is crucial to help the developers optimize internet search results.

Usually, a web application is divided into two separate parts: *frontend* and *backend*. The frontend is the view layer, where the user interacts with a browser using the page elements by clicking, selecting, or typing. This is where the data from the server is received, formatted, and displayed to the user through the browser.

The backend is generally the larger part of a web application. It comprises the business logic, communication handlers, the majority of the data processing, and the data storage. Data storage employs a separate database management system connected to the backend.

The frontend and backend communicate with each other. Data requests are forwarded by the frontend to the backend, and the data returned by the backend is received, formatted, and displayed by the frontend to the user.

In our simple example of fetching the latest transaction in a user’s bank account, the action is interpreted by the frontend of the application, which is running in the browser on the user’s desktop. This request is then sent through the internet to the backend of the application, which validates whether the user is allowed to perform the action, fetches the data, and returns the list of transactions back to the frontend loaded in the browser. The browser then formats and displays the data to the user.

Application Programming Interface (API)

No software is useful without communicating with internal and external components. So how can the client communicate with the web application? How can the frontend send data to the backend?

Software applications communicate with each other via an *application programming interface* (API), running over basic internet communication *protocols*. Protocols are standards and rules developed to ensure that two or more applications exchange commands and data.

The main benefit of an API is to decouple different parts of application while allowing them to cooperate in processing data. APIs also centralize the data flow in pre-defined channels, acting like a gateway that ensures that everyone uses the same way for coming and going. In web applications, APIs are vital for the application's functionality, as they allow user interaction, the delivery of processed information, requests for data storage, and many other tasks. An API can be used by the client to request actions that will be executed on the server, for example.

Let's go back to the banking web application example. To log into an account through a web application, the user generally types data such as username and password into certain text fields and clicks a "Login" button. The browser grabs this information and calls a backend API. The web application running on the remote server receives the user data, validates the user, verifies the right of the user to gain access, and finally sends a response back to the browser. In order for the user to communicate with the server, it is mandatory for both client and server to send data back and forth. That is what APIs enable.

Notice that the bank's web application does not expose other sensitive information; it shows the user only the fields that are allowed and necessary for a desired interaction. The rest is hidden from the user.

Communication between APIs can be based on very different designs and protocols. The *hypertext transfer protocol* (HTTP) is by far the most frequently used protocol in web applications. *Hypertext* is text with links to other texts, the concept underlying the links in HTML web pages. Hyperlinks thus form the basis for constructing web pages and displaying them in browsers.

HTTP was designed for client-server applications, where the resources are requested from a server and then returned to the client over the network using a predefined structure specified by the HTTP protocol.

For a structured web application, software engineers design the application with separate parts or modules. This separation of responsibility allows clearly defined tasks and responsibilities, leading to faster development and better maintenance.

Let’s take, as an example, an application with two internal modules: one that implements the business logic, and the other relying on a third-party integration. This third party is an external company that provides its API for some specific purpose — weather forecasting, say. If the weather server is down, it is impossible to get weather details, and if this data is crucial to the final processed output, the user might experience some temporary headaches if there is no alternative source for the data.

Now imagine that this third-party provider is replaced and the new one has a different way of handling the same API. The separation of modules means that developers of only one module have to update it. The business logic in the other module doesn’t have to be touched at all, or at least calls for minimal updates.

The need for clear process structures also influences the design of the APIs in order to make them easier to use. The *representational state transfer* (REST) concept is an architectural style with a set of guidelines for designing and implementing access to data in an application.

There are six REST principles. For the sake of simplicity, we are going to explain three that are most relevant to this lesson:

Client-server decoupling

The client should know only the resource URI, through which communication with the server occurs. This principle allows greater flexibility. For example, when the backend side of the application is refactored, or there is a major architectural change to a backend database, the frontend does not need to be updated in conjunction. It simply continues to send the same HTTP requests to the backend.

Statelessness

Each new request is independent from the previous ones. It is no coincidence that the HTTP protocol is widely used for applications that follow REST principles, since HTTP has no knowledge of previous HTTP requests; for each new request, all the necessary information must be sent in order to correctly process the request. For example, a web application that implements this principle has no knowledge whether the client is logged in (authenticated). So for each HTTP request, the client must send an authentication token. The server can use this token to verify whether the request should be blocked or processed.

One of the main advantages of this principle is easier scaling, because the server can process millions of requests without checking user details.

Layered architecture

The application is composed of multiple layers, and each layer can have its own logic and purpose, such as security or data acquisition. The client may never know how many layers exists, or whether they are communicating directly with a specific layer inside the application.

APIs following the REST principles are called *RESTful*, and in the modern web, the REST design is followed by many web applications. Although a RESTful API does not need to implement these principles using the HTTP protocol, it is almost universally used in the REST model given its robustness, simplicity, and ubiquitousness in the world wide web environment.

Architecture Types

Dozens of architectural styles and standards exist that attempt to organize the structures of web applications, Like almost everything in the computer world, there is no “winner,” only a set of pros and cons for each model. An important paradigm is the so-called *microservice architecture*, that was created as an alternative to the older *monolithic architecture*.

The microservice architecture is a software model composed of multiple interdependent *services* that together form the final application. This architecture aims to decentralize the codebase: Multiple layers of software are split into multiple smaller applications for better maintenance (Microservice architecture).

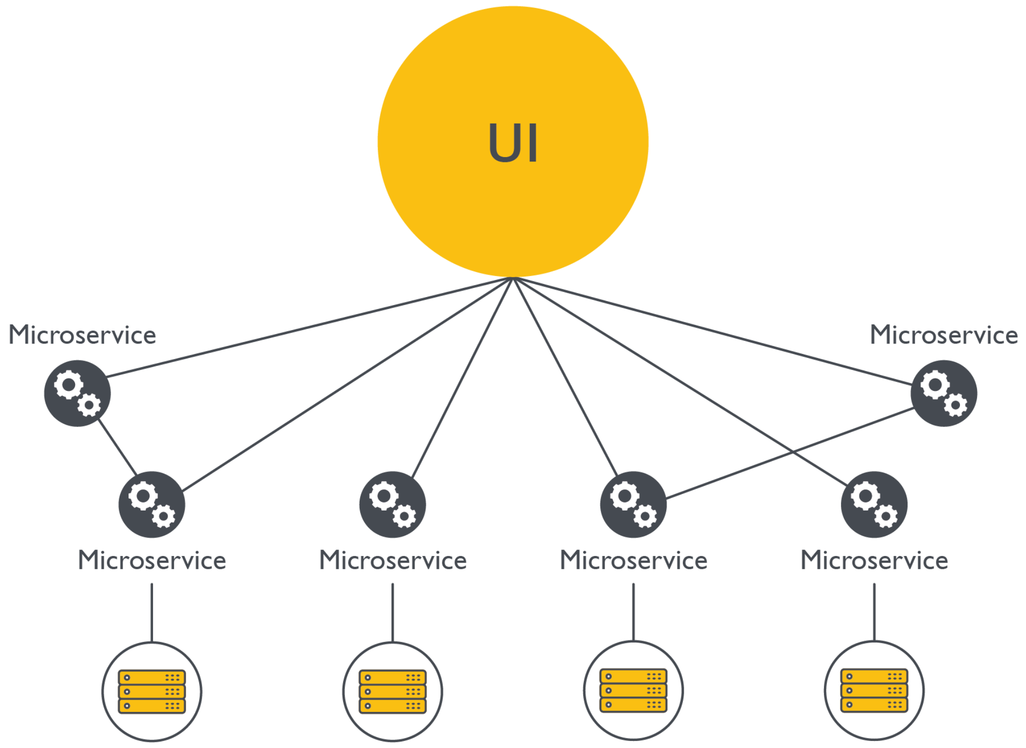


Figure 2. Microservice architecture

In contrast, a monolithic architecture contains all services and resources of the application in one application (Monolithic architecture).

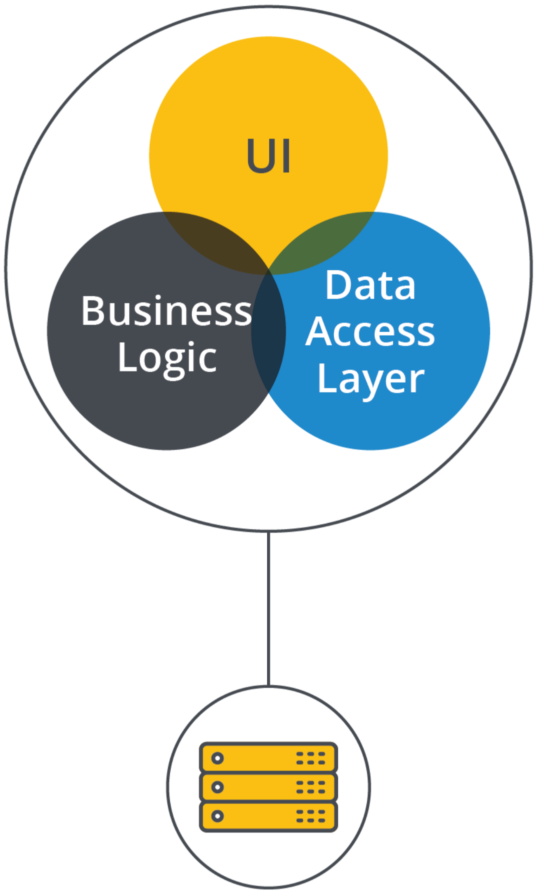


Figure 3. Monolithic architecture

The images show that the microservice model is decentralized and the application relies on multiple services, each with its own database, codebase, and even server resources. As the name implies, each microservice should be smaller than its monolithic counterpart, which takes responsibility for all services.

The monolithic application encapsulates all its resources into one single unit. All the business logic, data, and codebase are centralized in one huge block, which is why it's called a "monolith."

Users hardly notice whether a web application is running as a monolithic or microservice model; the choice should be transparent. Our banking application, for example, could be a monolith where all the business logic regarding payments, transactions, loans, etc. is located in the same codebase running on one or more servers. On the other hand, if the banking appliction uses a microservice style, it probably has a microservice dedicated to processing payments and another microservice just for issuing loans. The latter microservice calls yet another microservice to analyze the probability that the applicant will default on the payment. The application could have thousands of smaller services.

The monolithic approach requires more maintenance overhead when the application grows larger, especially with multiple teams coding in the same codebase. Given the centralized software resources, it is highly likely for one team to change something that breaks the another team’s part of the application. This could be a real headache for larger teams, especially when there is a great number of teams.

Microservices are much more flexible in that regard, because each service is managed by only one team. One team can, of course, manage more than one service. The code changes are easily done and competing resources are not a real issue. Since each service is interconnected, any point of failure could have a negative impact on the entire application. Furthermore, since there are multiple database instances, servers, and external APIs communicating with each other, the resilience of the whole application is only as good as its weakest microservice.

One advantage of the monolithic approach is having one centralized data source, which makes it easier to avoid data duplication. The approach also reduces the consumption of cloud resources, because one larger server needs fewer computer resources than multiple decentralized servers. A microservice application of roughly the same size puts a greater burden on the cloud.

Guided Exercises

1. What are the main differences between a fat and a thin client?

2. Is it correct to assume that every website is a web application?

3. What is the REST model?

4. What is the preferred model for developing large and modern web applications with multiple development teams? Why?

5. What is the most commonly used protocol to exchange data between web applications?

6. Name two disadvantages of multi-page applications compared to single-page applications.

7. Describe one advantage of a monolithic system over a microservice system, and one advantage the microservice system has compared to a monolithic one.

Explorational Exercises

1. In 2021 the NASA Perseverance Rover landed on Mars, with one of its goals to determine whether life ever existed on Mars. Although the Rover could be controlled by distance here on Earth, it also can control itself in most situations. Why is it a good idea to project a rover like that as a fat client?

2. Consider a modern, self-driving, personal car that connects to an external server to exchange data. Should it be a fat or a thin client?

Summary

This lesson explained the core concepts of software architecture for web applications. The lesson explained how they are commonly structured and organized, and the main differences between the monolithic and microservice models. We covered the concepts of servers and clients, and the basics of web application communication between clients and other software programs.

Answers to Guided Exercises

1. What are the main differences between a fat and a thin client?

   A fat client does not require a constant connection to a remote server that gives back critical information to the running client. The thin client relies heavily on the information processed by an external source. Another difference is that a fat client is responsible for the bulk of the data processing, thus requiring more computing resources than its thin counterpart.

2. Is it correct to assume that every website is a web application?

   No. There are websites that are not software applications. A web application interacts with the user, who might input data and use web functionalities in real time. Simple websites such as an advertisement for a social event, which functions like a web banner, are not web applications. These non-interactive web sites are easier to maintain and require tiny computing resources to host and deliver the web pages. A web application requires much more computing resources, more robust servers, and functionalities that handle users, such as restricted access and permanent data storage.

3. What is the REST model?

   The REST model is a software architecture model providing applications with a development guide for better usability, clarity, and maintainability. One of the principles outlined in the set of REST guidelines is the *layered architecture*, used primarily for cohesion and to lower the dependency of the various APIs' internal components.

4. What is the preferred model for developing large and modern web applications with multiple development teams? Why?

   The microservice software model provides a flexible framework in which teams collaborate on the same software application, providing easier concurrency for two or more teams maintaining a large web application. Because the framework is decentralized, each team can update a specific business domain without having to update other components.

5. What is the most commonly used protocol to exchange data between web applications?

   HTTP is the most used protocol for exchanging data and commands between servers and clients.

6. Name two disadvantages of multi-page applications compared to single-page applications.

   A multi-page application reloads all the elements in the web page when the user triggers some actions, instead of updating only the changed elements. Performance suffers from this design. Another disadvantage of an MPA is clunkier user interactivity, where each page load creates a loss in user friendliness. In contrast, the visual effect could be continuous in an SPA.

7. Describe one advantage of a monolithic system over a microservice system, and one advantage the microservice system has compared to a monolithic one.

   A monolithic system can make data administration easier because the data is located in one big database, instead of being scattered in multiple databases. A microservice application, on the other hand, can improve code development and maintenance; multiple teams can work on different business logic without blocking the progress of other teams.

Answers to Explorational Exercises

1. In 2021, the NASA Perseverance Rover landed on Mars. One of its goals was to determine whether life ever existed on Mars. Although the Rover can be controlled over a long distance by an application on Earth, it also can control itself in most situations. Why is it a good idea to design a rover like that as a fat client?

   The time a communication signal takes to be sent from Earth and received on Mars can vary depending on the positions of those planets, but it can take up to twenty minutes. Thus, command and control of a distant rover in motion is impossible, especially taking unexpected situations into account. Ideally, the rover should command itself in most situations. That is achieved using artificial intelligence (AI) training (machine learning), so that the rover becomes more independent from manual commands. To make that possible and to rely less on distant signals, the rover was projected to have its own resources and the majority of the computing processes ran locally, matching the definition of a fat client.

2. Consider a modern, self-driving, personal car that connects to an external server to exchange data. Should it be a fat or a thin client?

   An autonomous vehicle could delegate the heavy data processing to an external and reliable server, but this would be susceptible to offline periods when critical data processing is required. Therefore, it is imperative for the autonomous vehicle to process the majority of tasks — and this requires it to be a fat client with multiple redundancy.

051.3 On-Premises and Cloud Computing

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 051.3

Weight

1

Key knowledge areas

- Understanding the concepts of on-premise and cloud computing
- Understanding common cloud operation models
- Understanding common types of cloud services
- Understanding the major benefits and risks of cloud computing and on-premise IT infrastructure

Partial list of the used files, terms and utilities

- Cloud computing
- On-premises IT infrastructure
- Data center
- Public, private and hybrid cloud
- Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS)
- Cost models
- Security
- Data ownership
- Service availability

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	051 Software Fundamentals
Objective:	051.3 On-Premises and Cloud Computing
Lesson:	1 of 1

Introduction

Everyone is talking about the cloud nowadays. Businesses are evaluating services from a range of vendors — including major companies such as Amazon.com and Microsoft — to see what the cloud can offer.

But the idea of cloud computing goes back only to the late 2000 decade. And the term is a vague one — so vague that many people, including the Free Software Foundation, discourage the use of the term.

However, the idea of cloud computing embodies some specific useful concepts, and is a crucial trend in modern computing. This lesson looks at what the cloud is and, at a high level, how it works both technically and financially. We'll look at benefits and risks of the cloud. Along the way, we'll see why this topic is related to open source.

On-Premises and Cloud Computing

If you work or study in an organization that has more than a couple computer systems, it almost certainly has a *data center*: a separate room to hold the organization's servers, usually locked and air-conditioned. An *on-premises* (or *on-premise*) data center is simply one that the organization maintains for its own use.

There are several alternatives to running a data center on-premises. For decades before the trend we call “cloud computing,” businesses would set up data centers that hosted computers on behalf of clients. Thus, you might license five servers from the business, and give the business various specifications for CPU, memory, and storage, and then upload your software to the servers. This business model is *remote hosting*.

Remote hosting offers many advantages. The organization can effectively outsource the administrative expertise of buying and setting up servers to the remote hosting business, which could get bulk purchases. In other words, you avoid the responsibility of having an on-premises IT infrastructure. The remote hosting business ensures physical security, also freeing the client from that worry. Finally, setting up a new server on a remote hosting business is much faster than buying, shipping, and setting up the server on-premises.

An organization can also maintain an on-premises data center while licensing more servers in the remote environment to recover from disasters or provide extra compute power during periods of heavy use.

Note that remote computing assumes the presence of a fast and reliable network. We’ll explore this issue along with other benefits and weaknesses in another section.

All the advantages of remote hosting apply also to cloud computing. Cloud computing is technically different because no particular physical computer is devoted to your organization. Instead the cloud vendor runs multiple systems for multiple clients on each physical system, using an extra layer of software known as *virtual machines*.

Thus, a cloud service runs a data center like any other organization, but serves other organizations instead of (or in addition to) itself. The data center stores thousands of physical computers. On each physical computer it runs an operating system (usually called a *hypervisor*) that supports multiple virtual machines. Each virtual machine can be spawned and deleted quickly. Each virtual machine supports one operating system run by a client (Cloud computing).

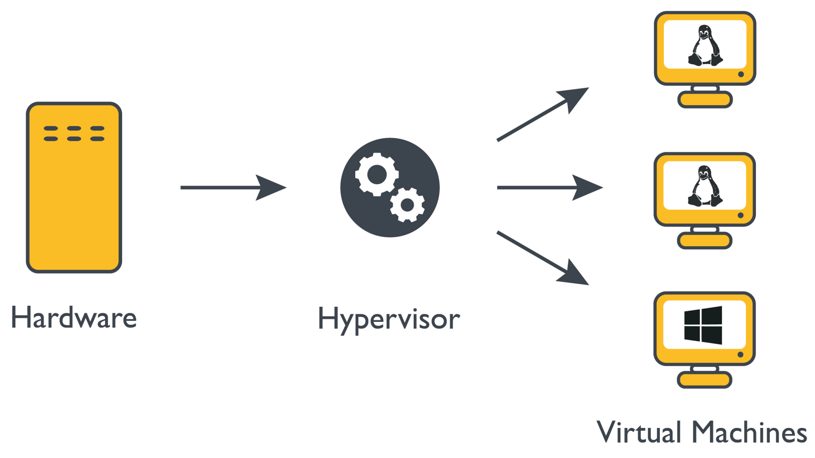


Figure 4. Cloud computing

Where does free software and open source come in? When a business is running huge numbers of computers and deploying operating systems on the fly, it’s important not to get bogged down dealing with licenses. Although there are licensing models for proprietary operating systems in the cloud, they are more complicated than simply running an open source virtual machine and operating system. Open source is usually cost-free, too.

The beginning of cloud computing is usually traced to Amazon.com’s launch of Amazon Web Services (AWS) in 2006. There are now dozens of cloud companies, including offerings from such big vendors as Microsoft, Google, Alibaba, and IBM. AWS is still the largest offering. The vendors compete fiercely on developing new features and services, because they are all strong on cost and reliability.

The advantages of cloud computing build on those of remote computing. Costs are lower because one physical system can run many servers for many clients, and can be kept busy constantly. A client that needs more computing power quickly for a spike in usage can spawn new systems in seconds. The systems can be managed automatically through an application programming interface (API). Again, we’ll take a closer look at benefits and risks later.

Common Cloud Operation Models

Before detailing operation models, it’s worth noting that many companies have adopted cloud models that totally change their way of programming and offering services. Instead of updating each application once or twice a year, the companies allow rapid updates. They can do this because the cloud allows them to shut down virtual machines and start up new ones almost instantly with the new version of the application. The organization can also scale up and down quickly, so they like to break applications into many modular parts, sometimes called *microservices*.

But in this section we’ll focus on everyday cloud models.

The cost model for the cloud is very different from on-premises costs. On-premises data centers require the one-time purchase of a server, along with routine costs for power, air conditioning, and administration. These disappear when you license systems from a cloud vendor. Instead, you are charged for what you use. Cloud vendors divide your computer usage into time periods and charge you for each period. They also charge by the amount of data you store on their systems.

So far, we’ve been talking about vendors who offer compute power to customers; this is called the *public cloud*. But there can also be a *private cloud*. Some large organizations run their own on-premises data centers like a cloud. They provide services only to their own departments or subdivisions, but they treat each of their departments like a client of a cloud vendor. The data center keeps track of how much compute time, data, etc. is used by each department and charge it for that usage.

Many organizations use multiple cloud services for various reasons, such as to protect against vendor failures, keep data in a certain geographic region, or take advantage of special features offered by a particular vendor. In addition, it’s common to maintain both an on-premises data center and servers in the cloud, a practice called *hybrid computing*.

A client who signs up for a cloud service can choose which geographic regions to run in. For instance, Amazon curently offers regions in the US West, US East, several regions in Europe, and more regions in every part of the world. Normally, you’d choose the region closest to you. But many organizations want to operate in multiple regions because they are international in scope. The organizations sometimes need to keep data in a particular place to adhere to Europe’s General Data Protection Regulation (GDPR), or China’s Personal Information Protection Law (PIPL).

Each region normally is further subdivided into *zones* or *availability zones*. Running in multiple zones is recommended in case a disaster causes one zone to go down.

Although the cloud is notable for sharing physical computers among multiple clients, some vendors can devote a single computer to a particular client who is worried about security. Because no other organization is using the computer, the client feels a bit safer running their sensitive services and uploading their data to the cloud. This option brings cloud computing closer to old-fashioned remote hosting.

Common Types of Cloud Services

At different levels, cloud computing looks very different and is aimed at different types of users.

Infrastructure as a Service (IaaS) is the category that system administrators usually deal with. IaaS provides just hardware and the software that supports virtual machines. It's up to system administrators of the client to load an operating system and their desired applications onto the virtual machine. The system administrator handles nearly everything the same way as on an on-premises data center.

Platform as a Service (PaaS) is a more recent invention, used mostly by programmers. Here, the programmer doesn't worry about the operating system and doesn't have to load the libraries that the program uses. All of that is provided by the cloud vendor. The programmer just uploads functions that run on the platform. A related concept is *serverless* computing.

Software as a Service (SaaS) is an application running on a cloud system. Every time you log into a social media site, order an item from an online store, visit a web page to enter your hours into a job tracking system, or enter a form on a government site, you are using SaaS. The bulk of the application is running on the remote system, and the only part of the application running on your computer is the web page displayed by your browser.

Database as a Service (DaaS) is often added to the preceding categories. A data service, Amazon's S3, was actually the first cloud offering. A DaaS offering can simply be an instance of a popular database server such as MySQL, Oracle, or MongoDB running in the cloud. Big cloud vendors also offer propriety databases that run only in their cloud offerings. In any case, you read and write the database as if you had it on your systems.

Other variations on those basic categories are offered by some companies, such as Security as a Service.

Major Benefits and Risks of Cloud Computing and On-Premises IT Infrastructure

Before examining cloud computing in detail, let's try an analogy. Running an on-premises data center is like buying a house. If the basement floods or the boiler stops working, you need to find someone to fix it. In contrast, remote hosting and cloud computing are like renting an apartment: The landlord is responsible for fixing the boiler. Furthermore, in the cloud you can quickly add and remove data loads, just as you can change apartments more quickly than you can change which house you own.

In an apartment, a landlord might even provide appliances and furniture. In our analogy, this is like the numerous services that cloud providers offer, such as databases and analytics.

Now we can look at the benefits and risks of using the cloud, instead of or in addition to your own data center.

Flexibility is probably the most compelling reason to move to the cloud. If you're a retailer who needs to run more servers near Christmas, or a tax preparation accountant doing most of your business at tax season, you'll want the cloud in order to spin up new servers at a moment's notice and then delete their virtual machines later.

Costs can be lower in the cloud for several reasons. You are sharing a physical server with many other applications, so the computers are used more efficiently. Because cloud vendors are large, they can achieve economies of scale in purchases, administration, cooling, and other infrastructure requirements. Finally, the clients are freed from many administrative tasks—although system administration is by no means going away. Clients still need system administrators to create and upload their software (known as *instances* in the cloud), to authorize users, and other tasks related to business operations. System administrators have to learn the vendor's API and the rules for using the service, a training cost you should factor into your plans.

On the other hand, you have to be careful how much use you make of the cloud. It can be hard to keep track of how much computer power you're using when you can quickly spin servers up, especially if you automate your scaling. You might find an unpleasantly large bill at the end of the period.

Is the cloud more carbon-efficient than running our own computers? Research finds that cloud vendors can run their systems much more efficiently than you or I can. But we have to communicate with those systems over a network, which requires a lot of electrity to power all the networking equipment. So unfortunately the cloud increases our carbon footprint.

Service availability is sometimes better with the cloud. Certainly, if you depend on your own on-premises data center, you are vulnerable to all kinds of problems ranging from natural disasters to malicious internal saboteurs. But data centers in the cloud also go down. So you should take advantage of the different availability zones and spread out your risk. There are tools that allow you to switch your services from a failed zone to a working one.

If you use the services offered by the cloud vendor, such as a database in the cloud, you are vulnerable to bugs in that service. Of course, you can also suffer from bugs in the software you load into your system.

A greater risk in using vendors' services is lock-in. You can usually find an automated conversion tool to move your data out of the vendor's system and into a new one, but the tool might not do a complete job.

Security can be better in the cloud because the vendor’s staff are probably more expert than your security staff. On the other hand, cloud vendors are big and well-known, furnishing obvious targets for attack. Also, adding an extra piece of software -— the hypervisor that controls the virtual machines -— introduces a new potential hazard. Researchers have found vulnerabilities in hypervisors.

Although the client remains the legal owner of its data, storing the data in the cloud theoretically leaves it more vulnerable. Usually, the client encrypts the data to protect it in case of a break-in. Privacy regulations, such as the previously mentioned GDPR, require data to be stored in a data center in a region considered to be safe.

Ultimately, most security attacks start at a high level, such as sending email with malware to an unsuspecting employee. It doesn’t matter whether you’re running on-premises or in the cloud. But a malicious intruder who takes over an employee’s account will not get much farther unless they can take advantage of vulnerabilities in your servers; again it’s not clear whether running in the cloud makes much difference because most vulnerabilities are found in the software rather than the cloud service.

Finally, consider your bandwidth and networking costs. Your customers, and probably your staff, are communicating with servers that might be hundreds of miles away. If the network connecting is unreliable or slow, the performance of cloud servers will be worse than your on-premises data center. But nowadays, everybody is connecting with remote workers, SaaS services, and other systems that are geographically removed. Your network performance will affect nearly everything you do, whether or not you’re in the cloud.

Guided Exercises

1. Why is a physical computer in a cloud center used more efficiently than a computer in a traditional on-premises data center?

2. What is a hybrid cloud?

3. What type of cloud computing most often requires work on the part of a system administrator at the client side?

4. How should you protect your service from going down if you use a cloud vendor?

Explorational Exercises

1. Compare the different kinds of costs you experience when running your servers in a cloud to the costs of running them on-premises.

2. You operate out of the Middle East, but have many customers in Europe and the Far East. Describe where you would place your services in a cloud offering.

Summary

This lesson outlined how cloud computing works and the trade-offs of using the cloud versus running systems in your own data center, on-premises. You learned different business and cost models, including the differences between public, private, and hybrid clouds. You also learned the different major types of cloud offerings and what each is used for.

Answers to Guided Exercises

1. Why is a physical computer in a cloud center used more efficiently than a computer in a traditional on-premises data center?

   In the cloud, each computer can run multiple instances of operating systems, and even run instances uploaded by different clients. Therefore, the computer is more often in use.

2. What is a hybrid cloud?

   A hybrid cloud uses both data centers at a cloud vendor and one or more on-premises data centers.

3. What type of cloud computing most often requires work on the part of a system administrator at the client side?

   Infrastructure as a Service (IaaS) requires the client to perform system administration for tasks such as creating and uploading instances of the operating system and applications.

4. How should you protect your service from going down if you use a cloud vendor?

   Choose several zones in each region where you run your service, because it’s highly unlikely that many zones will fail simultaneously.

Answers to Explorational Exercises

1. Compare the different kinds of costs you experience when running your servers in a cloud to the costs of running them on-premises.

   In a cloud, you pay for your CPU use and data storage for each period measured by the vendor. But you don’t pay any hardware costs. On-premises, you have the fixed cost of the hardware, along with other equipment such as air conditioning, plus recurring costs such as power and physical maintenance.

2. You operate out of the Middle East, but have many customers in Europe and the Far East. Describe where you would place your services in a cloud offering.

   Use a Middle Eastern region for your own offices and Middle Eastern clients. A region in Europe is important in order to comply with the General Data Protection Regulation (GDPR). You might need a region in China to comply with China’s Personal Information Protection Law (PIPL). In any case, having Far Eastern and European regions is valuable for better performance when interacting with customers in those places.

   Within each region, choose several zones to protect against the failure of a single zone.

Topic 052: Open Source Software Licenses

052.1 Concepts of Open Source Software Licenses

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 052.1

Weight

3

Key knowledge areas

- Understanding the definitions of open source software and free software
- Awareness of other kinds of monetarily-free software
- Awareness of important events in the history of open source
- Understanding what a license is and what rights licenses commonly manage
- Understanding how existing software can be used to create derivative works
- Understanding license compatibilities and incompatibilities
- Understanding dual licensing and conditional licensing
- Understanding consequences of license violations
- Understanding the principles of copyright law and patent law and how they are affected by open source software licenses

Partial list of the used files, terms and utilities

- Free Software Foundation (FSF) free software definition
- Open Source Initiative (OSI) open source software definition
- Licenses
- Contracts
- Public domain software

- Freeware
- Shareware
- License stewards
- Permissions to use, modify and distribute code and software
- Derivative works and code reuse
- Closed source / proprietary software
- Paid distribution
- Modified and unmodified software distribution
- Hosting software as a paid service
- License compatibility
- Dual and multi licensing
- Conditional licensing
- Software patents
- Explicit and implicit patent license grants

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	052 Open Source Software Licenses
Objective:	052.1 Concepts of Open Source Software Licenses
Lesson:	1 of 1

Introduction

Any software developer appreciates when a problem has been efficiently solved before. If the solution is available online, the understandable reflex is to try it out: to copy or link the existing source code into one's own code base, test it, and leave it there if it works — and then to forget about it.

But some caution is in order. More often than not, software source code available on the internet is covered by a free and open source (FOSS) *license*. This chapter will go into some basic concepts of FOSS from a legal standpoint and why it is a good idea to not take FOSS source code for granted, but rather to closely observe the licensing conditions.

As a basis for understanding your legal obligations in software, understand that a license governs almost all software. A license is a kind of contract. All software, including web sites, should be distributed with a written version of its license (often called "terms and conditions"). Some programs and web sites make you check a box saying that you have read their terms and conditions (and you should, although most people never do). In any case, you implicitly accept the license by using the software.

Definitions of Open Source Software and Free Software

Free and open source software has been around for quite some time, and the terms are often seen together. However, there is some discrepancy between what some declare to be “free software” and the formal meaning of *free software*, as well as *open source software*. Spoiler: “Open source” does not just mean that anyone can look at the source code — that would be “source available” software. Free and open source software is much more than that.

Free and open source software is contrasted with other kinds that are considered *proprietary* or *closed source* because they don’t offer all the freedoms discussed in this lesson.

Perhaps the most cited definition of free software is:

> Think of “free speech,” not “free beer.”
>
> — Richard Stallman, Selling Free Software

We’ll unpack the fertile brew in that statement during the following sections.

Free as in Speech: True Freedom for Users

On the one hand, developers can decide to simply forego the rewards and difficulties of selling their software, and just give it away without compensation. In English this software is “free” in the sense that no one has to pay for it, but free only like beer that is given away. This no-cost distribution says nothing about further licensing conditions, and leaves the user with the obligation to always take a closer look: They might not have been granted the necessary rights to change or further distribute the software, for example.

On the other hand, developers can opt to make their software “free software” in Stallman’s sense. This term (which he coined in the 1980s) refers to the user’s essential freedoms summarized as follows:

1. To run the software
2. To study it
3. To give it to others (*redistribute* it)
4. To redistribute copies of the modified versions

“Free software” in this definition is championed by the Free Software Foundation (FSF) which introduced the term *copyleft* (which has no legal meaning, but presents a philosophical consideration) to characterize the license on free software.

Open Source Software

Out of the free software movement, open source advocates arose in the late 1990s as a way to make free software easier to understand and more popular among people outside the movement. A nonprofit foundation, the *Open Source Initiative* (OSI), was founded in 1998, and Linus Torvalds, initial author of the Linux kernel, gave his support to the concept. The Open Source Initiative formalized the *Open Source Definition* to include the following criteria:

1. *Free redistribution*: One is free to decide how to redistribute the program, whether cost-free or for sale, so long as no royalty or license fee is demanded. This freedom includes incorporating the program into another program.
2. *Source code availability*, be it online or provided along with the software.
3. Allowing the creation and distribution of *derived works* and modifications.
4. *Integrity of the author's source code*: Modifications can be restricted if recipients are allowed to modify the program through patches at build time and to distribute these patches along with the source code. The distribution of software *built* from modified source code may not be restricted.
5. *No discrimination against persons or groups*: For instance, a license with permission for use of the software by "teachers only" would not satisfy the Open Source Definition.
6. *No discriminations against fields of endeavour*: For instance, do not restrict commercial use.
7. *Distribution of license*: Everyone who receives the program has the same, original license.
8. *License must not be specific to a product*.
9. *License must not restrict other software*: For instance, other software bundled with this software could have a different license.
10. *License must be technology-neutral*.

Freedom vs Open Source

The Free Software Foundation did not endorse the term "open source," insisting that it hides the key goal of freedom. Thus, while advocates of free software and open source software seem to be pursuing and advocating for the same concept, the movements have separate motivations. Stated in a simplified form, free software advocates emphasize the rights of developers and users, whereas open source advocates prioritize the software's widespread use and success.

Pretty much all free software qualifies as open source, while there are many licenses that are considered to be open source (as approved by OSI) but are not free according to the FSF.

Because the differences between free and open source concern goals and motivations more than the content of the licenses, and since both terms remain in frequent use, many advocates refer to both definitions together using the phrases free and open source software (FOSS) or *free/libre and open source software* (FLOSS). The term “libre” refers to freedom in many Romance languages.

Other Kinds of Monetarily Free Software

In addition to the broad categories of FOSS and proprietary software, a range of other distribution strategies exist. Some of these strategies are:

Shareware

This term generally refers to proprietary software that is for sale, but can be used for free with limited functionality until the user decides to buy the full version.

Freeware

This term describes software that is distributed free of cost and without limitation on use, but not necessarily in comformance with a formal definition of the free software. Freeware in many cases is proprietary and the source code is often not released at all.

Source available software

Sometimes the developers of proprietary software make their source code available (to facilitate better bug reports, for instance), but impose acquisition of a proprietary license as a condition on the use of the source code in other projects. This kind of availability with strict limitations should not be confused with actual open source software.

Shared source software

This term was introduced by Microsoft in 2001, when the company decided to make some of their software source code available online for research and testing. Don’t confuse this narrow definition with shareware or source available software.

Public-domain software

This is software on which the authors have waived all copyrights. This definition does not apply in all jurisdictions (especially with those where the author is granted rights of “droit d’auteur” or “author’s rights,” as in France or Germany). Licenses such as the “Unlicense” have been introduced, which are meant to have the same effect. Moreover, software may also enter the public domain when the duration of copyright has expired.

Principles of Copyright Law and How They Are Affected by Open Source Software Licenses

First and foremost: If there is no license information available for a certain source code file or project, you cannot assume that the file or project lacks copyright protection. Indeed, the contrary is the case, at least since most nations worldwide signed the *Berne Convention* since 1887.

In this treaty, the signatory nations agree that a literary or artistic work is protected by copyright as soon as it exists (or in other words, as soon as it is “fixed” into a medium). That means that an author does not have to register or apply for copyright. They still can do so in some countries, and registration might be necessary for infringement actions in some jurisdictions including the US.

Also, the signatories agree to respect the copyright of any author of another signatory country, which, as of November 2022, amounted to 181 of the 195 countries in the world.

However, not everything ever created is copyright protected. In order to qualify as a *work* protected by copyright, the creation needs to satisfy some basic criteria: For example, facts and ideas cannot be protected by copyright, but a text explaining an idea may be protected if it reaches a certain degree of *originality*. The measure of originality differs around the world, but in many cases, the barrier for protection is very low. Many jurisdictions are currently considering how much artificial intelligence can be used to create a work deserving of originality and hence copyright.

Depending on the jurisdiction, computer programs are protected by copyright as literary works: That is, copyright does not apply to the idea or algorithm but to its implementation in source code.

Copyright gives the author the exclusive rights (among others) to copy, modify, sublicense, distribute, and publish the work. Reception of the work is free, so one does not need a license to read a book or to listen to a song on the radio, as long as doing so does not require making a permanent copy.

Because the author’s rights come into being without the need for registration, anyone who wants to copy, modify, sublicense, distribute, or publish the work of another author has to get permission first. This is where licenses come into play, as contracts between the author of a work and a person who wants to exercise some of the exclusive rights of the author.

FOSS licenses are offered to anyone without a fee. Anyone can create their own license and try to get it approved by the OSI or FSF. But using an existing FOSS license is highly recommended because of their general acceptance and the familiarity that knowledgeable software users have with the contents of the license and its obligations. In fact, only a handful of the many licenses approved by the FSF or OSI are in common use.

Principles of Patent Law

In contrast to copyright, which does not protect ideas, patents protect inventions (ideas) without the need for the idea to be fixed (yet) in a machine or process. A further difference is that inventors must apply for patents explicitly and register them with the patent office of the country where protection is sought.

Without diving too deeply into patent law and its requirements, we'll just point out a very central issue: Patent protection requires (among other criteria) an idea with a certain technical aspect. Historically, the ideas for a new meal recipe or a new board game do not qualify for patent protection, whereas the ideas for a new cooking machine or a game console might qualify.

In the context of computer programming, the question arises whether software could be subject to patent protection. This depends on both the legal jurisdiction and the particular application: For example, in Germany, computer programs as such are generally excluded from patent protection. However, if programs are combined with a physical object — e.g., when software controls an automated braking system in a car — patent protection can be sought for the application including the physical element of the brake and not the software as such, and might therefore qualify.

In other jurisdictions, such as the US, patents may be granted for computer programs as such, depending on case law developments.

The concept of patent protection should be kept in mind in FOSS licensing. Some licenses (such as the GPLv3) allow patents on the FOSS software by explicitly granting permission for the software's use. But some licenses do not even mention patents (such as the BSD-3-Clause license) and yet other licenses explicitly exclude them (such as the Creative Commons licenses). However, under certain circumstances, patent grants may be implied by the license or may be read into the license.

NOTE The different FOSS licenses are covered in subsequent lessons.

Especially when dealing with embedded software, such as software in audio devices, special attention should be paid to patents connected with the software, e.g. by conducting patent checks on authors of the software.

Licensing Contracts

As mentioned before, licenses are contracts between the author of a work and someone who wants to excercise some of the exclusive rights of the author. Some of the more extensive FOSS licenses, such as the GNU General Public Licenses version 2 and version 3, include provisions for contract termination. FOSS licenses always include rights grants regarding the central freedoms.

Usually, the following rights are stated or can be read into the license text:

> …the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software…
>
> — MIT license

License stewards are persons or, in many cases, organizations such as the FSF, who manage versions of the licenses. For example, section 9 of GPLv2 declares the FSF to be the license steward:

> The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.
>
> — GNU General Public License version 2

Some license stewards also publish frequently asked questions (FAQs) to help answer questions regarding the licenses. These can be useful as a conversation starter between licensee and licensor.

Distribution

Distribution of software (in binary or source code form, in particular) is a central aspect of most FOSS licenses, since mere *use* of FOSS software usually does not trigger any licensing obligations:

> Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted…
>
> — GNU General Public License version 2

Most licensing obligations arise only with distribution, i.e., passing on a modified or unmodified copy, be it on a tangible medium such as a CD or via download. In some cases, licensing conditions are also triggered if software is running on a server (i.e., without ever distributing the source code) while the user interacts with the software.

Distribution of executable software might require the delivery of the source code, license text, and copyright notices as well, depending on the FOSS license type. Some licenses require modification notices to be put in the source code if modified code is being distributed.

Distribution could also trigger copyleft effects; i.e., upon distribution of GPLv3 licensed software that has been modified, the source code of the entire modified software may have to be released under the GPLv3.

While FOSS may be sold, usually no licensing fees may be imposed. This means, for instance, that someone can sell software under the GPLv3 license as just a binary, but since the source code is available (or has to be offered), people interested in the software can always opt to obtain the sources (perhaps from somebody else, for free) and to build the software from the sources.

Derivative Works

When one set of developers incorporate code into their own project from someone else's project, the result might be a *derivate* or *derived* work. The details vary from one project to another and from one license to another, and because the details require some sophistication with software development techniques, we'll just say that the developers have to make sure they are incorporating the code in a way that conforms to the license.

The most important issue regarding derivative works is that, for some licenses such as the GPL, a derivative work must be released under the same license. Such licenses are called *reciprocal*.

The immediate practical significance of a reciprocal requirement is that, if you use GPL'd code in your own project in a way that makes yours a derivation, you have to reveal your source code and let others build products on it. This licensing requirement is highly desired by some developers who want to encourage more people to use free licenses. However, the license reduces the appeal of the GPL for some developers who might potentially use free code.

Many other free and open source licenses do not impose that requirement. These tend to be called *permissive* licenses.

Consequences of License Violations

If, at the time of distribution, licensing conditions are not met (e.g., if GPLv3 licensed software distributed as a binary is not accompanied by an offer of source code), the licensing contract is violated. The consequences of licensing violations depend on the license. Violation of the GPLv3 license, for example, may lead to termination of the license. Any further actions that require permission by the author, e.g. distribution of the software, then constitute copyright violations.

If a company includes GPLv3 licensed software in a product and violates the license, a claim may be filed requiring the company to recall their products. Intentional copyright violations can even lead to criminal charges.

Some licenses have specific clauses allowing the licensee to fix the violation within 30 days of notification. If the person distributing the software manages to comply with the license within this period, the license is reinstated.

License Compatibility and Incompatibility

Large software projects often include software under different licenses, each license specifying its individual requirements. Such projects can come up against hurdles using software that requires its license to be used on derivative works. This is because the licensing terms of such copyleft licenses may differ, rendering them incompatible. Publishing or distributing a software project that integrates components under different copyleft licenses might not be possible without violating one of the licenses.

Some copyleft licenses explicitly list compatible licenses making it easier to use a copyleft-licensed component under another copyleft license.

Most permissive licenses are compatible with other licenses. For instance, MIT-licensed components may be used in a GPLv3 licensed project without risking licensing violations. However, a GPLv3 licensed component may not in every case be used in a MIT licensed project without violating the terms of the GPLv3. Compatibility might therefore not always work both ways.

Before a software project is put into the market, developers and their legal advisors should conduct a thorough license compatibility check, to avoid running into licensing violations. Open source compliance management should be integrated into the early stages of a software development process to avoid delays in software distribution caused for example by license incompatibility issues. Search the source code thoroughly for applicable licenses (e.g., by using software scan tools) and check whether all licensing conditions are met.

Dual Licensing and Multiple Licenses

Some software may be available under several licenses. For example, a licensor may choose to *dual license* their project under both a copyleft license such as the GPLv3 and a proprietary license. The proprietary license might be required, for instance, if the potential licensee incorporates the code into their own proprietary product. Each developer determines whether they can adhere to the GPLv3 conditions or must obtain the proprietary license, and most likely entail a licensing fee.

As pointed out earlier, some software may include components under various licenses with differing licensing conditions. Although most licenses do not often cause incompatibility, different licenses may come with different requirements for different parts of the software.

Guided Exercises

1. What does the acronym FOSS stand for?

2. Which of the following explicitly belong to the user's essential freedoms for free software?

Run software	
Study software	
Copy software	
Change software	
Publish software	
Redistribute software	

3. Is source code found on the internet with no licensing information free for modification and distribution by anyone? Please explain.

4. Can software be patented?

Yes	
No	
It depends	

5. What is a license steward?

The same as a licensor	
Someone who may propose future versions of a license	
Someone who may terminate a license	
Someone who manages software of airplanes	

6. Does license compatibility always work both ways?

Yes, if two licenses are compatible, it does not matter whether A is included in B or B is included in A.	
No, in some cases license A may be included in a project under license B, but B may not be allowed to be distributed under license A.	
No, license compatibility is always unidirectional.	

Explorational Exercises

1. Are the GPLv2 and the LGPLv2.1 license compatible? Please give a short explanation.

2. Can software be published under Open Content licenses (such as CC-BY)?

Yes, CC licenses can be applied to any copyrightable work.	
No, software may only be protected by patents.	
No, CC licenses do not apply to software.	

3. Why is distribution important in the context of FOSS licensing?

Summary

This lesson provides an introduction to the basic concepts of free and open source software as well as the underlying concepts of copyright and patents. It explains some of the foundations and history of both free software and open source software and helps distinguish both of those from proprietary licensing concepts. The Open Source Definition by the OSI helps categorize licenses as open source licenses.

While dozens of FOSS licenses already exist, anyone is free to introduce additional licenses.

The concept of licensing is not to be underestimated: Licenses provide permissions to deal with software in ways otherwise reserved exclusively to the author. Breaches of licenses can result in legal disputes. Legal advice regarding license compliance should therefore be sought before software is distributed or incorporated into other project code.

Answers to Guided Exercises

1. What does the acronym FOSS stand for?

   Free and Open Source Software

2. Which of the following explicitly belong to the user’s essential freedoms for free software?

Run software	X
Study software	X
Copy software	
Change software	X
Publish software	
Redistribute software	X

3. Is source code found on the internet with no licensing information free for modification and distribution by anyone? Please explain.

   No. Source code, if it reaches the threshold of originality, is by default copyright protected as a literary work. Since modification and distribution constitute exclusive rights of the author, no one but the author may do so or give permission to do so.

4. Can software be patented?

Yes	
No	
It depends	X

5. What is a license steward?

The same as a licensor	
Someone who may propose future versions of a license	X
Someone who may terminate a license	
Someone who manages software of airplanes	

6. Does license compatibility always work both ways?

Yes, if two licenses are compatible, it does not matter whether A is included in B or B included in A.	
No, in some cases license A may be included in a project under license B, but B may not be allowed to be distributed under license A.	X
No, license compatibility is always unidirectional.	

Answers to Explorational Exercises

1. Are the GPLv2 and the LGPLv2.1 license compatible? Please give a short explanation.

   If software A is licensed under the GPLv2 and software B is licensed under the LGPLv2.1, it is possible to use B in A, since LGPLv2.1 allows for use under the conditions of the GPLv2 license. However, A cannot be used in B under the terms of the LGPLv2.1. If A is being used in B, the entire software has to be licensed under the GPLv2, which is possible, because LGPLv2.1 allows for the software's use under the GPLv2.0.

2. Can software be published under Open Content licenses (such as CC-BY)?

Yes, CC licenses can be applied to any copyrightable work.	X
No, software may only be protected by patents.	
No, CC licenses do not apply to software.	

3. Why is distribution important in the context of FOSS licensing?

   Many FOSS license obligations are triggered upon distribution of the software. If software is never distributed, but only changed and used in a closed system (such as a division of a company), it may be possible to use the software without adhering to the licensing obligations.

052.2 Copyleft Software Licenses

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 052.2

Weight

3

Key knowledge areas

- Understanding the concept of copyleft software licenses
- Understanding the rights granted by copyleft software licenses
- Understanding the obligations created by copyleft software licenses
- Understanding the main properties of common copyleft software licenses
- Understanding the compatibility of copyleft software licenses with other software licenses
- Awareness of the terms “reciprocal license” and “restrictive license”

Partial list of the used files, terms and utilities

- Copyleft
- Distributing
- Conveying
- Tivoization
- GNU General Public License, version 2.0 (GPLv2)
- GNU General Public License, version 3.0 (GPLv3)
- GNU Lesser General Public License, Version 2 (LGPLv2)
- GNU Lesser General Public License, Version 3 (LGPLv3)

- GNU Affero General Public License, Version 3 (AGPLv3)
- Eclipse Public License (EPL), version 1.0
- Eclipse Public License (EPL), version 2.0
- Mozilla Public Licence (MPL)

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	052 Open Source Software Licenses
Objective:	052.2 Copyleft Software Licenses
Lesson:	1 of 1

Introduction

The importance of licenses for both the use and the development of software has already been described. It is therefore not surprising that free software has also been characterized by new approaches to licensing from the outset: Conditions for the unrestricted use or collaborative development of software must be legally defined in order to be protected and enforced.

Aware that copyright law, which is firmly anchored in almost all legal systems worldwide, could not be questioned or replaced, software developers took an approach as early as the 1980s that respects copyright regulations but supplements them with new regulations that emphasize the principle of “freedom”: *copyleft*.

Copyleft and the GNU General Public License (GPL)

Richard Stallman, then a developer at the renowned MIT, founded the *GNU Project* in 1983 to develop what he considered to be a “free” operating system. It soon became clear that the code developed by the project had to be legally protected so that it could not simply be taken over by commercial providers and thus become “non-free.”

Stallman therefore founded the non-profit *Free Software Foundation* (FSF) in 1985, which summarizes its mission on its website as follows: “The Free Software Foundation is working to secure freedom for computer users by promoting the development and use of free (as in freedom) software and documentation…”

A key instrument for this mission is a license that respects applicable law (especially copyright law) on the one hand and implements its own ideas of freedom in a legally clean manner on the other. The result was the first version of the *GNU General Public License* (GPLv1) in 1989. This license and numerous articles — such as “What is Free Software?”, written by Stallman in 1992 — make clear the motivations and values of free software developers, who now also see themselves as a “movement.”

The programmatic core is still formed by the “four essential freedoms” formulated by Stallman in the aforementioned article, the numbering of which begins with 0:

> - The freedom to run the program as you wish, for any purpose (freedom 0).
> - The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
> - The freedom to redistribute copies so you can help others (freedom 2).
> - The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.
>
> — Richard Stallman, What is Free Software?

Unlike licenses for commercial products, which place restrictions on use in the foreground, free software is about maximum freedom for users and developers.

The preamble of the GNU GPLv1 summarizes this as follows:

> Specifically, the General Public License is designed to make sure that you have the freedom to give away or sell copies of free software, that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.
>
> — GNU General Public License, version 1

This means that everyone has the right to use, distribute, and modify software under the GPL without restriction (which is possible because the source code is accessible, i.e. “open”) and in turn to distribute the modifications. It is even possible to charge money for passing on the software:

> Actually, we encourage people who redistribute free software to charge as much as they wish or can. If a license does not permit users to make copies and sell them, it is a nonfree license….Free programs are sometimes distributed gratis, and sometimes for a substantial price. Often the same program is available in both ways from different places. The program is free regardless of the price, because users have freedom in using it.
>
> — Free Software Foundation, Selling Free Software

But if the freedoms are so far-reaching, to what extent is software protected under this license, for example from being incorporated into proprietary products?

This is the role of the previously mentioned copyleft principle, which the GPL already applies in version 1 even if the term does not yet appear explicitly:

> You may not copy, modify, sublicense, distribute or transfer the Program except as expressly provided under this General Public License.
>
> — GNU General Public License version 1

This means that all these freedoms are linked to the condition that users preserve these freedoms in everything they do with the software.

Copyleft therefore not only guarantees freedoms, but also requires all users to grant these freedoms to others. This is achieved by stipulating that software under a copyleft license (such as the early GPLv1) may be modified and redistributed only if the modifications are published under the same conditions, i.e. under the same license.

The ideal of free software, namely the collective use and further development of software, therefore takes precedence over the personal needs that individuals might have in relation to the software. The principle of reciprocity is crucial: Those who use freedoms must also grant them. Copyleft licenses are therefore often referred to as *reciprocal*.

This completely new approach to a software license already proved to be legally sound and practicable in version 1 of the GPL, so the GPL has undergone only two major revisions in the almost 40 years during which the modern IT market has developed.

GPLv2 and GPLv3

In 1991, the Free Software Foundation presented version 2 of the GNU General Public License (GPLv2), which established itself as the most popular license for free software projects over many years. For example, the core of the Linux operating system is still licensed under GPLv2 today.

Compared to version 1, version 2 is primarily concerned with more precise definitions to avoid ambiguities. For example, version 2 explains in much greater detail what is meant by “source code.”

Also of interest is the new Section 7, which sets the principle of freedom and thus the validity of the license as absolute and does not allow any compromises — for example, the integration of less free parts into the software:

> If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.
>
> — GNU General Public License version 2

It was not until 16 years later, in 2007, that the FSF published a new version of the GPL in order to take account of technical innovations — such as the provision of software services via the internet — as well as issues of compatibility with other FOSS licenses. However, the license remains stable in terms of its core statements and merely adds details for further clarification. Let’s take a closer look at some of these additions.

While the GPLv2 still generally refers to the provision of software as *distribution*, GPLv3 specifies this process with two new terms: *propagation* and *conveying*. The main reason for this is that the term “distribution” is defined in numerous copyright laws worldwide. To avoid ambiguity or conflict, the GPLv3 chooses these new terms and defines them as follows:

> To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.
>
> To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.
>
> — GNU General Public License version 3

With the significantly growing number of commercial software products whose distribution is restricted by manufacturers through technical measures such as registration codes or hardware components (so-called *dongles*), there were a number of international legal initiatives in the late 1990s to criminalize the circumvention of these measures. This so-called *digital rights* *management* (DRM), also derogatorily referred to by opponents as *digital restrictions management*, is a thorn in the side of the FSF, as the measures fundamentally contradict the demand for the free distribution of software.

In response, version 3 of the GPL contains a passage stating that software under the GPL may not be modified with reference to legal DRM requirements. This also means that software licensed under GPLv3 may use DRM, but that others are also permitted to circumvent such measures.

The term *tivoization* is also frequently used in this context. The word appeared explicitly in the first drafts of GPLv3 but was not included in the final version. The term goes back to the company TiVo, which used GPLv2-licensed software in its digital video recorder, but at the same time technically prevented modified software from being installed and used on the device. In the FSF's opinion, this contradicted the principles of the GPL, and after some discussion, the GPLv3 takes this into account with a paragraph on so-called *user products*. It generally stipulates that products that use GPLv3-licensed software must also provide information on how this software can be modified.

A further addition concerns *patents*, which the FSF fundamentally rejects on software with reference to their obstruction of freedom and innovation. This is already stated in the preamble to GPLv3:

> Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.
>
> — GNU General Public License version 3

The license text also contains several passages that allow the inclusion of code under a patent by means of a “non-exclusive, worldwide, royalty-free patent license” from the licensor in order to protect users of such code from disputes between patent holders and licensees.

The GNU Affero General Public License (AGPL)

With the increasing availability and speed of the internet, more and more services are emerging in which the software is merely installed on the servers of the vendor — the *Application Service Provider* (ASP) — and whose customers interact with the services via the internet. The trend has gained the name *Software as a Service* (SaaS).

In such cases, the GPLv2 did not provide any clarity as to whether and how the source code (possibly modified by the provider) should be made available. Version 3 of the GPL closes this loophole, known as the *ASP loophole*, by explicitly referring in section 13 to another license issued by the FSF in 2007: the *GNU Affero General Public License* version 3 (GNU AGPLv3). The name goes back to the company Affero, which developed and published the first two versions of this license.

This AGPLv3 basically corresponds to the GPLv3, but explicitly regulates the ASP problem in the section "remote network interaction." Furthermore, both licenses explicitly state that they can be combined with each other without restriction.

In summary, the GNU AGPL is a complementary supplement to the GPL. The AGPL extends the scope of the GPL by applying copyleft to software that is no longer used in local installations, but exclusively in the form of services transmitted via networks.

Compatibility of Copyleft Licenses

The development of free software thrives on building on the work of others, i.e. integrating, modifying, and sharing the source code of others. If all parts of a modified or newly compiled software are under the same copyleft license, for example the GPLv3, this is possible without any legal problems. The license simply requires that the result is distributed under the same license.

Things become more complicated when software consists of parts that are licensed under different licenses. Several factors need to be taken into account here.

Combined and Derivative Works

Free software is sometimes created under very different conditions. Changes range from simple error corrections to complex projects with millions of lines of code. Regardless of the scope, a basic distinction is made between two types of works when it comes to licensing: *derivative* and *combined*.

Let's assume, for example, that software project A is missing a certain functionality. Instead of developing this functionality from scratch, it makes sense to combine code from another project B, which offers exactly this functionality, with A. The software from B would not even have to be changed for this, but could simply be added to A. It is therefore a combined work. If both A and B are under the same copyleft license, there are no problems for the combined work.

If A and B are under different copyleft licenses, caution is required: Is the combination of A and B already a separate work? And, if so, under which license can or must it be licensed? Or can a conflict be avoided by ensuring that both parts A and B remain separate with their respective licenses and do not constitute a new work?

It becomes even more difficult with derivative works, i.e. when project A can make use of B's functionality only by incorporating code from B directly into the code of A. This integration creates a new, derivative work whose parts can no longer be separated.

Copyleft comes into play for a derivative work. If, for example, A is under a copyleft license such as the GPLv3, the new, derivative work must also be under the GPLv3 in accordance with the principle of reciprocity. But what if B is under a different license? Is it a copyleft license, and is it compatible with the GPLv3? Or, if it is a different type of license, can B also be published under a different license, i.e. relicensed? Or are the licenses of A and B even categorically mutually exclusive?

The aim in this lesson is not to list the possible combinations and possible solutions. The important thing is to illustrate the complexity of the problem resulting from the combination of very different issues.

Technically, the first thing to clarify is how the different parts of the software (in our example A and B) work together: Can they be separated from each other or are there processes whose execution can no longer be clearly assigned to one part or the other?

From a legal point of view, questions arise about the relationship between the licenses of A and B. Are they compatible with each other, or only in parts or only in one direction? Is relicensing an option?

We can only hint at the complexity of these questions here, not resolve them. Such decisions call for sound legal knowledge. Thus, licensing decisions for new works, but especially for combined and derivative works, should be made *early*, *carefully*, and always after detailed *legal advice*.

Weaker Copyleft

The copyleft has proven to be extremely robust over the past decades, especially in the form of the GPL in versions 2 and 3. However, special technical requirements have prompted the FSF to react by adapting its licenses. The GNU Affero General Public License is one such example. We discuss other examples in the following subsections.

The GNU Lesser General Public License (LGPL)

A method frequently used in software development is the use of small modules for standard tasks, such as opening or saving files. These modules — or collections of such modules — are referred to as *software libraries*. They are usually not independently executable applications, but routines that the actual application integrates as required. The integration process is known as *linking*, whereby a distinction is made between two methods.

With *static libraries*, the actual application (via auxiliary programs and intermediate steps) firmly integrates the code of the modules into the executable file of the application. With *dynamic libraries*, on the other hand, the application integrates a module only when required at runtime and loads it into the working memory.

This raises a question with regard to copyleft and licensing: What are the licensing implications of linking? For example, does this form of taking over code require an application that uses a library under the GPL to adopt the GPL itself? And does this apply to both static and dynamic linking?

The linking process is so ubiquitous in software development, and the issues associated with reciprocal copyleft so extensive, that the FSF looked early on for a licensing solution that permitted linking through the *GNU Lesser General Public License* (LGPL). The LGPL is updated at the same time as each new version of the GPL. The name of the license in version 1 was the *GNU Library General Public License*, which revealed the original problem that led to the license's creation. In versions 2 and 3, "Library" replaced by "Lesser" to indicate what is actually at stake, namely a pragmatic weakening of the copyleft principle.

A library licensed under the LGPL can therefore be used by software without this software itself automatically being subject to the copyleft. Software projects under so-called *permissive* licenses (covered in another lesson), in particular, benefit from this compromise, as it creates legal protection for the combination of approaches that are not per se compatible under licensing law.

Any changes to LGPL-licensed software are still subject to the copyleft, i.e. they must also be under the LGPL.

However, the LGPL-licensed library must be interchangeable, to allow the user of the software to replace the library with a modified version. Appropriate conditions must be created for this, i.e. information must be provided on how such a replacement can be made.

Other Copyleft Licenses

Other FOSS projects and organizations also look for the best legal framework for their needs, and hence develop their own licenses. One example is the *Mozilla Foundation*, founded in 1998 and today best known for the two projects it supports: the Firefox internet browser and the Thunderbird email client.

Version 1 of the *Mozilla Public License* (MPL) was published in 1998, and the current (as of 2024) version 2.0 in 2012.

Like the LGPL, the MPL is often referred to as a "weak copyleft" license. In fact, it seeks to strike a balance between the strict requirements of the copyleft and the possibilities for integration with commercial products. It achieves this, among other things, through a principle known as *file-level copyleft*: If you make a change to a file that belongs to software under the MPL, you can integrate this file into proprietary software as long as the modified file itself is remains under the MPL and is therefore accessible.

Another example of a weak copyleft license is the *Eclipse Public License* (EPL) from the Eclipse Foundation. The current version 2.0 from 2017 is very similar to the MPL and is often referred to as the most “business-friendly” copyleft license. However, the various copyleft licenses — and there are others besides the ones mentioned here — often arose due to historical developments rather than clear legal differences.

Guided Exercises

1. What does the abbreviation GPL stand for?

2. Why are copyleft licenses also referred to as reciprocal?

3. Which FSF copyleft license is suitable for software libraries?

4. Which English terms replace the term “distribution” in GPLv3 and why?

5. Which of the following copyleft licenses were issued by the Free Software Foundation?

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

Explorational Exercises

1. Which of the following are copyleft licenses?

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

2. Can one typically create a derivative work combining parts of two software projects that are under different strong copyleft licenses? Give reasons!

3. Which of the following licenses have a strong copyleft and which have a weak copyleft?

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

4. Describe some compatibility issues that could arise when combining software under a weak copyleft license with software under a non-copyleft license.

Summary

This lesson deals with the characteristics of software licenses that follow the principle of copyleft. Developed in the 1980s by the Free Software Foundation, the GNU General Public License (GPL) is currently the most popular license with a strong copyright. Despite several revisions up to the current version 3, its basic requirements have remained virtually unchanged: The freedoms granted by the license to use, redistribute, and modify the software without restriction must be preserved at all times. This means that the modified software may be distributed only under the same conditions (i.e., the same license).

Technical innovations as well as the desire for legal leeway when collaborating with other projects have prompted the FSF to develop supplementary or alternative licenses. The GNU Lesser General Public License (LGPL), for example, takes into account the static or dynamic linking of software libraries frequently used in software development. The GNU Affero General Public License (AGPL) responds to the technical trend toward using software in the form of services via the network (especially the internet), i.e. not in local installations.

In addition to the FSF, other projects such as the Mozilla Foundation and the Eclipse Foundation have developed copyleft licenses. These also seek a compromise between securing the freedoms granted by the license and a simpler relationship to software under other licenses by means of a weaker copyleft.

In principle, license compatibility is an important issue with copyleft licenses, because the principle of free, collaborative software development must be reconciled with the legal conditions determined by the respective license. In any form of combination of software under different licenses, the legal conditions must be examined carefully in each individual case.

Answers to Guided Exercises

1. What does the abbreviation GPL stand for?

   General Public License

2. Why are copyleft licenses also referred to as reciprocal?

   The principle of copyleft requires that freedoms granted by a license must also be granted to others without restriction. For example, if you make a change to software under the GPL, you are obliged to make these changes available to others under the same conditions, in accordance with this reciprocity.

3. Which FSF copyleft license is suitable for software libraries?

   GNU Lesser General Public License (GNU LGPL)

4. Which English terms replace the term “distribution” in GPLv3 and why?

   The terms “convey” and “propagate” replace “distribution.” The background to this is that the term “distribution” is firmly anchored in international copyright law. To avoid conflicts or misunderstandings, the GPL in version 3 does not use this term.

5. Which of the following copyleft licenses were issued by the Free Software Foundation?

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

Answers to Explorational Exercises

1. Which of the following are copyleft licenses?

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

2. Can one typically create a derivative work combining parts of two software projects that are under different strong copyleft licenses? Give reasons!

   No. Licenses with a strong copyleft generally require that modified versions are under the same license. This also excludes relicensing. The combination of licenses, when both follow these principles, therefore represents an irresolvable contradiction.

3. Which of the following licenses have a strong copyleft and which have a weak copyleft?

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. Describe some compatibility issues that could arise when combining software under a weak copyleft license with software under a non-copyleft license.

   While a strong copyleft requires the modified software to be distributed under the same license, the conditions are “relaxed” in the case of a weak copyleft. Nevertheless, any combination with other licenses raises fundamental questions of compatibility. Important aspects need to be considered when answering these questions: Are the original parts of the two software projects clearly separate in the new work and, if necessary, can they still be licensed differently? At what level does the connection between the two original software projects actually take place? Is source code directly adopted or is it “only” dynamically linked against the other software (library)? Does one of the two licenses generally allow relicensing? All questions of this kind can be answered only after a precise analysis of the respective licenses and with specialist legal expertise.

052.3 Permissive Software Licenses

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 052.3

Weight

3

Key knowledge areas

- Understanding the concept of permissive software licenses
- Understanding the rights granted by permissive software licenses
- Understanding the obligations created by permissive software licenses
- Understanding the main properties of common permissive software licenses
- Understanding the compatibility of permissive software licenses with other licenses

Partial list of the used files, terms and utilities

- 2-Clause BSD License
- 3-Clause BSD License
- MIT License
- Apache License, version 2.0

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	052 Open Source Software Licenses
Objective:	052.3 Permissive Software Licenses
Lesson:	1 of 1

Introduction

Permissive licenses are currently the most widely used open-source licenses, in contrast to *restrictive* licenses such as the GNU General Public License (GPL). Permissive software licenses tend to be simple and flexible, and provide a wide range of freedom to their authors — perhaps the widest freedom available among open source licenses.

This type of license grants broad freedom to software developers regarding the use, modification, and redistribution of the software, so long as they acknowledge the original author. For example, someone can usually distribute a derivative work under a closed-source license, so long as the work includes attribution for the author of the work from which the new one was derived.

The principle behind this logic is to allow the maximum possible dissemination of the software. Permissive licenses claim to benefit individuals and the public by facilitating commercial use of the software, while preserving the rights to the original author through attribution.

It is no coincidence that the first and most important permissive software licenses were developed in the academic field, such as the BSD-like licenses of the University of Berkeley in California or the MIT/X11 License of the Massachusetts Institute of Technology. These are known as academic open source licenses. Their influence within the industry was such that they laid the groundwork for similar permissive software licenses conceived outside of academic contexts, such as the Apache License from the Apache Software Foundation.

Rights and Obligations of Permissive Software Licenses

In general, the most popular permissive software licenses grant the licensee the unlimited right to:

Use the software

Software covered by a permissive software license can be used by anyone (from individual users, to commercial companies, to public authorities) and for any purpose, whether personal or professional.

Modify the software

The software can be improved, adapted, or even integrated as a component (e.g., a library) into other software.

Redistribute the software

If the software is modified, taking the form of a derivative work, it can be redistributed under different licenses, including proprietary ones.

The only obligation in permissive software licenses usually is that of attribution: The licensee is required to indicate the name of the original author of the software in the derivative work and include a copy of the license text in any redistribution of the software.

Features of the Most Important Permissive Software Licenses

This section explains the differences between the MIT/X11 license, the most common BSD licenses, and the Apache 2.0 license, all of which are in wide use nowadays.

MIT/X11 License

The MIT/X11 license (text: https://opensource.org/license/mit), also known as the X11 license, is one of the academic licenses mentioned earlier. The license takes its name from the X Window System software developed by the Massachusetts Institute of Technology back in 1987. This license is one of the oldest and most popular permissive software licenses, partly because of its simple, clear language.

This license grants the licensee the rights to:

- Use, modify, and distribute the software
- Commercialize the software, with or without modifications
- Release derivative works under a different license, even as closed-source software

The sole obligation of the licensee is to include the copyright notice and the license text in the source code of the software or of its derivative works.

The MIT/X11 license is considered to be compatible with all the most important copyleft licenses, both weak and strong. In particular, the MIT/X11 license is compatible with the

- GNU General Public License (GPL), versions 2.0 and 3.0
- GNU Lesser General Public License (LGPL), versions 2.0 and 3.0
- Mozilla Public License (MPL)

That means that derivative works of software originally licensed under the MIT/X11 license can be redistributed under one of the above-mentioned licenses or be included in projects released under one of the above-mentioned licenses.

2-Clause BSD License

The 2-Clause BSD License (text: https://opensource.org/license/bsd-2-clause) derives from the original BSD license, created in 1980 by the University of Berkeley in California as the license for BSD, its Unix-like operating system.

The license is well known for its simplicity and, as its name indicates, consists of just two short clauses. It is substantially identical to the MIT/X11 license. Like the MIT/X11 license, this one requires attribution in the software. Additionally, the 2-Clause BSD License requires the licensee to include the license text in the documentation and other materials provided when redistributing the code.

To summarize, the 2-Clause BSD license grants the rights to:

- Use, modify, and distribute the software
- Commercialize the software, with or without modification
- Release derivative works under a different license, even as closed-source software

The licensee has the obligation to:

- Include the copyright notice and the license text in the source code and binary of the software or of its derivative works

- Include the copyright notice and the license text in documentation or other materials provided with the software

The 2-Clause BSD license is considered to be compatible with all major copyleft licenses, both weak and strong. In particular, the 2-Clause BSD license is compatible with:

- GNU General Public License (GPL), versions 2.0 and 3.0
- GNU Lesser General Public License (LGPL), versions 2.0 and 3.0
- Mozilla Public License (MPL)

Therefore, derivative works of software originally licensed under the 2-Clause BSD license can be redistributed under one of the above-mentioned licenses or be included in projects released under one of the above-mentioned licenses.

3-Clause BSD License

The 3-Clause BSD license (text: https://opensource.org/license/bsd-3-clause) is yet another variant of the original BSD license, consisting of just three clauses. The main feature distinguishing this license from the sister 2-Clause BSD license is the "non-endorsement clause," which aims to prevent the name of the original author from being exploited to distribute or sell derivative works.

The 3-Clause BSD license grants the licensee the rights to:

- Use, modify, and distribute the software
- Commercialize the software, with or without modifications
- Release derivative works under a different license, even as closed-source software

The licensee has the obligation to:

- Include the copyright notice and the license text in the source code and binary of the software or of its derivative works
- Include the copyright notice and the license text in documentation or other materials provided with the software
- Refrain from citing the name of the copyright holder or contributors to endorse or promote products derived from this software, without specific prior written permission

In order to explain the latter obligation — the "non-endorsement clause" — let's think of a scenario in which a company or individual creates a derivative work starting from software licensed under a permissive software license. In the absence of this clause, the licensee could promote the new work by indicating that it is a derivative of software by a well-known developer, in order to benefit from their prestige.

As with its sister license, the 3-Clause BSD license is considered to be compatible with all the most important copyleft licenses, both weak and strong. In particular, the 3-Clause BSD license is compatible with:

- GNU General Public License (GPL), version 2.0 and 3.0
- GNU Lesser General Public License (LGPL), version 2 and 3
- Mozilla Public License (MPL)

Therefore, derivative works of software originally licensed under the 3-Clause BSD license can be redistributed under one of the above-mentioned licenses or be included in projects released under one of the above-mentioned licenses.

Apache 2.0 License

The first Apache license was developed by the Apache Software Foundation, a nonprofit organization established in 1999, to distribute its software and make it easy to include in other projects. The mission was to guarantee collaborative software development, embracing the open-source philosophy with a business-friendly perspective.

The Apache 2.0 license (text: https://www.apache.org/licenses/LICENSE-2.0), probably the most widespread among the permissive software licenses, differs from the licenses discussed previously in a few relevant aspects. The main aspect concerns the attribution of patent rights to each licensee, including a clause related to the termination of the patent license, so as to limit and prevent any claims for damages for patent infringement.

Two other obligations are significant. The licensee must release, under the same Apache 2.0 license, any parts or components of the software not modified by the licensee. Furthermore, the licensee must place prominent notices in any modified files stating that the licensee changed those files.

The Free Software Foundation, which promotes restrictive licenses, recommends Apache 2.0 as the best of the permissive software licenses for distributing small amounts of software and libraries.

The Apache 2.0 license grants the licensee:

- The right to use, modify and distribute the software
- the right to commercialize the software, with or without modifications

- The right to release derivative works under a different license, even as closed-source software
- A patent license to use, sell, import, and otherwise transfer the software covered by a patent
- A patent license to use, sell, import, and otherwise transfer the software which might otherwise infringe a contributor's patent claim

The licensee has the obligation to:

- Place prominent notices in any modified files stating that the files have been modified
- Release all unmodified parts of the original software under the Apache 2.0 license
- Include the copyright notice and the license text in the source code of the software and its derivative works
- Include the copyright notice and the license text in documentation and other materials provided with the software

The Apache 2.0 license is considered to be compatible with only some of the most important copyleft licenses. In particular, the Apache 2.0 license is compatible with:

- GNU General Public License (GPL), version 3.0 but not 2.0
- GNU Lesser General Public License (LGPL), version 3.0 but not 2.0
- Mozilla Public License (MPL), version 2.0 but not version 1.1

That means that derivative works of software originally licensed under the Apache 2.0 license can be redistributed under one of the above-mentioned compatible licenses or be included in projects released under one of the above-mentioned compatible licenses.

The limited compatibility between the Apache 2.0 license and other open source licenses is primarily due to the presence of clauses related to the grant of a patent license to the licensee.

Permissive Software Licenses in Relation to Other Open Source Licenses

Now that we have an overview of the common and essential features of permissive software licenses, we can move on to examine how they differ from the public domain software and from copyleft licenses.

Comparison to Public Domain Software

The first distinction between permissive software licenses and public domain software release lies in their very existence. The public domain is not an actual license, but just a way to release software. In other words, by definition, works in the public domain have no license.

The next distinction lies in the obligations that permissive licenses impose on the licensee. In fact, the user of a public domain release has no obligations whatsoever. However, anyone who uses, modifies, or redistributes software under a permissive software license must comply with the attribution obligation explained earlier: a requirement to include the name of the original author and a copy of the license text in the software.

The consequence of the attribution obligation is that no derivative work originating from software under a permissive software license can be released as public domain, because this would violate (or at least circumvent) the original author's right to receive attribution.

Comparison to Copyleft

The distinction between permissive software licenses and restrictive, copyleft licenses is more complex, especially considering the differences between various copyleft licenses. As seen in previous lessons, one major difference separates *strong* copyleft licenses (including GNU General Public License (GPL), version 2.0 and 3.0) from *weak* copyleft licenses (including GNU Lesser General Public License (LGPL), version 2.0 and 3.0, and the Mozilla Public License).

The major difference between restrictive licenses and permissive licenses lies in the copyleft principle, which is core to restrictive licenses. In conformance with this principle, the licensee who modifies software under copyleft license, and then distributes it, must necessarily release — totally or partially — the derivative work with the same license as the original software. Failure to comply with this obligation results in copyright infringement, with its attendant legal consequences.

On the contrary, permissive software licenses do not impose such an obligation. Those who use or plan to release software under this type of licenses can freely decide the license for their derivative work, including proprietary licenses.

Strong Copyleft

The difference between permissive software licenses and strong copyleft licenses are more marked than with weak copyleft licenses.

The essential difference is that strong copyleft licenses require derivative works to be released under the same license as the original software. This also applies if the copyleft-licensed software is integrated into another project: The entire derivative work must be released under the same strong copyleft license.

Weak Copyleft

Unlike strong copyleft licenses, weak ones have only a partial requirement to release a derivative work under the same license. To be more specific, the licensee who redistributes software released under a weak copyleft license must apply the same license to the portion of their work derived from the original one. For example, a library based on a library licensed under the LGPL must also be licensed under the LGPL.

The weak copyleft licenses were conceived precisely to bring their features closer to those of the permissive software licenses. They are nevertheless distinguished from permissive software licenses in that the latter do not impose, under any circumstances, an obligation to maintain the same license on a derivative or integrated work.

Guided Exercises

1. What are the two obligations generally imposed by permissive software licenses?

2. Which of the following licenses is *not* a permissive software license?

Apache 2.0 license	
LGPL license	
MIT/X11 license	
3-Clause BSD	

3. What distinguishes a permissive software license from a copyleft license?

Software released under a copyleft license cannot be distributed, whereas softare under permissive software license can.	
The works derived from software under a copyleft license cannot be released under a proprietary license, whereas software under a permissive software license can.	
Copyleft licenses are legally recognized only in the United States of America, whereas permissive software licenses are globally recognized.	

4. Which permissive software license grants a patent license to use, sell, import, and otherwise transfer the software covered by a patent?

Apache 2.0 license
2-Clause BSD license
MIT/X11 license
3-Clause BSD license

5. When software is released under the MIT/X11 license, can you distribute it under a proprietary license and sell a derivative work based on the software?

Explorational Exercises

1. You are modifying software distributed under the Apache 2.0 license and redistributing the derivative work under a proprietary license. What steps should you follow to be compliant with the Apache 2.0 license obligations?

2. Name at least three examples of popular projects released under permissive software licenses.

3. Why can’t you distribute, under a LGPL 2.0 license, software that includes components that were originally released under the MIT/X11 license, the Apache 2.0 license, and the 2-Clause BSD license? What different weak copyleft license can be used to release the software?

Summary

In this lesson you have learned: * What permissive software licenses are, the rights they grant, and the obligations they provide * The differences between permissive software licenses and other open source licenses * Features of the most popular permissive software licenses * Compatibility of permissive software licenses with other open-source licenses

Answers to Guided Exercises

1. What are the two obligations generally imposed by permissive software licenses?

   The obligation to indicate the name of the original author of the software and to include a copy of the text of the license in the derived work.

2. Which of the following licenses is *not* a permissive software license?

Apache 2.0 license	
LGPL license	X
MIT/X11 license	
3-Clause BSD	

3. What distinguishes a permissive software license from a copyleft license?

Software released under a copyleft license cannot be distributed, whereas softare under permissive software license can.	
The works derived from software under a copyleft license cannot be released under a proprietary license, whereas software under a permissive software license can.	X
Copyleft licenses are legally recognized only in the United States of America, whereas permissive software licenses are globally recognized.	

4. Which permissive software license grants a patent license to use, sell, import, and otherwise transfer the software covered by a patent?

Apache 2.0 license	X
2-Clause BSD license	
MIT/X11 license	
3-Clause BSD license	

5. When software is released under the MIT/X11 license, can you distribute it under a proprietary license and sell a derivative work based on the software?

Yes.

Answers to Explorational Exercises

1. You are modifying software distributed under the Apache 2.0 license and redistributing the derivative work under a proprietary license. What steps should you follow to be compliant with the Apache 2.0 license obligations?

   You should:

   - Insert into each modified file a notice attesting that the files have been modified.
   - Release all unmodified parts of the original software under the Apache 2.0 license.
   - Include the copyright notice and the license text in the source code of the software or of its derivative works.
   - Include the copyright notice and the license text in documentation and other materials provided with the distribution of the software.

2. Name at least three examples of popular projects released under permissive software licenses.

   - Angular web framework — MIT/X11 license
   - Ruby on Rails — MIT/X11 license
   - Apache HTTP Server — Apache 2.0 license
   - Kubernetes — Apache 2.0 license

3. Why can't you distribute, under a LGPL 2.0 license, software that includes components that were originally released under the MIT/X11 license, the Apache 2.0 license, and the 2-Clause BSD license? What different weak copyleft license can be used to release the software?

   Even though the MIT/X11 license and 2-Clause BSD license are compatible with the LGPL 2.0 license, the Apache 2.0 license is not. Software that includes components released under MIT/X11 license, Apache 2.0 license, and 2-Clause BSD license can be released under LGPL 3.0 license, because it is compatible with all those permissive software licenses.

Topic 053: Open Content Licenses

053.1 Concepts of Open Content Licenses

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 053.1

Weight

2

Key knowledge areas

- Understanding types of open content
- Understanding what constitutes content that is subject to copyright
- Understanding derivative works of copyrighted materials
- Understanding the need for open content licenses
- Awareness of trademarks

Partial list of the used files, terms and utilities

- Documentation
- Images
- Artwork
- Maps
- Music
- Videos
- Hardware designs and specifications
- Databases
- Data streams

- Data feeds

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	053 Open Content Licenses
Objective:	053.1 Concepts of Open Content Licenses
Lesson:	1 of 1

Introduction

Think about this hypothetical situation: Frank is a writer who wants to make some of his stories available to anyone on the internet, but under certain conditions. He wants his work to be free of cost. He wants anyone to be able to use the stories without asking for permission, but he doesn't want his work to be used commercially. Finally, he wants people to credit him properly if the stories are used for any purpose. He would like to do this without involving himself in complicated legal processes. We'll see later why a writer might want to do these things.

Emma is a filmmaker who teaches some students how to produce short films. Among other things, the students need a story to create their works. Emma knows Frank's stories and thinks they are perfect for her class.

What can Frank do to allow Emma to use his stories? This situation and many more can be solved by using an *open content licensing* model. Nowadays, millions of copyright-protected works can be reused by anybody without explicit consent from the copyright holder or the payment of a license fee, thanks to their distribution under open content licenses. Resources such as Wikipedia, Flickr, OpenStreetMap, Unsplash, and Jamendo are some examples of the many platforms that use this license model.

According to a very broad definition, *open content* refers to any work (including, for example, movies, music, images, texts, databases, data sets, documentation, maps, and hardware designs) whose free use and redistribution is permitted under copyright law. This also includes originally protected works whose term of protection has expired and which are therefore in the public domain. A somewhat narrower definition presupposes that a work has been explicitly placed under an open content license by its author, which allows use and distribution of the work without any payment or permission. The open content model is therefore not in opposition to copyright law, but complements it.

Copyright Fundamentals

Because open content licenses are based on copyright, you need to learn some aspects of copyright law in order to understand why the licenses are needed and what they allow.

Copyright is a part of a legal category known as *intellectual property*. Copyright protects original works by granting its creators exclusive legal rights to control certain uses of their works by other people and corporations. Generally, this means that no one else can copy, distribute, publicly perform, adapt, or do almost anything else other than view or read the work without the permission of the copyright holder.

The fundamentals we will examine in this section include what is copyrightable, along with who controls the rights and can grant permission to reuse a copyrighted work as Emma wants to do in our example.

As explicitly stated by the World Intellectual Property Organization:

> Copyright protects two types of rights. Economic rights allow right owners to derive financial reward from the use of their works by others. Moral rights allow authors and creators to take certain actions to preserve and protect their link with their work. The author or creator may be the owner of the economic rights or those rights may be transferred to one or more copyright owners. Many countries do not allow the transfer of moral rights. Copyright only protects the expression of facts or ideas. It does not allow the copyright holder to own or control the idea exclusively. For example, an illustration can be copyrighted but not the idea that originated it.
>
> — World Intellectual Property Organisation, Understanding Copyright and Related Rights

Economic rights protected by copyright last a long time, generally decades after the creator dies. Moral rights never expire.

Copyright grants rights to creative works, such as literary and artistic creations that must meet a certain standard of originality. The work must be a creation of its creator and not copied from another work. (There are ongoing debates about how much artificial intelligence can go into a work and still have it be considered original.)

Copyright protects only the expression of facts or ideas. It does not allow the copyright holder to own or control the idea exclusively. For example, an illustration can be copyrighted, but not the idea that originated it.

Copyright is automatic when a work is created and fixed in some tangible form, for example, a digital artwork or a song. This means that the rights are granted to the creator without even formally registering the work.

The people who promote open content licensing want to promote a free culture and the development of a digital commons. They have found the copyright system too restrictive and rigid for both users and creators. By creating easy-to-use standard licenses, the open content advocates simplify the use and distribution of works protected by copyright.

What Can Be Copyrighted?

Copyright laws vary from country to country. However, there are international agreements to standardize copyright laws. Original literary and artistic works can be copyrighted: for example, works in art, music, photography, film, television, literature, and programming. The particular rules to decide what is copyrightable, and how original a work is, vary depending on the region.

Sometimes these categories can be very general and apply to works that have both creative and strictly functional elements: For example, a short film is considered artistic work. However, portions of it can be uncopyrightable if they do not meet the standard of originality.

Derivative Works

Let’s continue with the hypothetical situation from the beginning of the lesson: Frank ultimately decided to publish his stories under an open content license so they can be used under the terms he chose. Since the stories came under that license, Emma used them in her class and decided to show some of her students' short films during a film festival. In this case, the content created by the students can be considered *derivative works*.

A derivative work or *adaptation* is a work based on existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, or any other form in which the original work is transformed or adapted. The derivative work becomes a second, separate work independent in form from the first. The transformation, modification, or adaptation of the work must be substantial in order to be original and thus protected by copyright.

Common Open Content Licenses Features

Every open content license asserts the copyright of the author and affirms that without a license from the author, any person using the work would be in violation of copyright. Therefore, such licenses work within the global copyright system instead of trying to overturn it.

Open content licenses also ensure that appropriate credits should be given to the author of the work. If recipients of the work distribute it to a third party, they should ensure that the original author is acknowledged and credited. Also, when recipients modify the work, the derivative work should clearly mention the author of the original and mention where the original can be found.

Unlike most copyright licenses, which impose restrictive conditions on uses of the work, open content licenses enable users to have certain freedoms by granting them rights. Some of these rights are common to virtually all open content licenses, such as the right to copy the work and the right to distribute the work. Depending on the license, the user may also have the right to modify the work, create derivative works, perform the work, display the work, and distribute the derivative works.

Open content licenses can control derivative works. These licenses normally include the right to create a derivative work and distribute it in any media. If a person licenses a painting under an open content license, the right to base another picture on it can be granted as well. These permissions are granted on the condition that others may use the derivative work freely, just like the original work.

Therefore, an open content license normally ensures that derivative works are licensed under the terms and conditions of the same open content license. But this obligation is not applicable when the work is included in a compilation. For example, if a person makes an album of songs, one of which is licensed under an open content license, not all the songs have to be licensed under the same terms.

Another important aspect of open content licenses is the control of commercial use of a work. People can license their works under an open content license while restricting the rights to noncommercial purposes. Alternatively, people may also grant all rights, including the right to use the work commercially.

Open content licenses don’t stop people from making money from their work. If a work is under a noncommercial license, a publisher or other commercial entity may publish the work by making an agreement with the copyright holders before doing so. In other words, even after releasing a piece of work on a noncommercial basis, authors may sell the copyright to a for-profit entity, provided that there is no exclusion placed upon continuing, noncommercial usage.

Importance of Open Content Licenses

What are some of the benefits of an open content model, and why would people use an open content license to distribute their creative work rather than relying on the traditional copyright model?

Open content licenses allow works to circulate more widely than if they were restricted by default. Therefore, new artists could benefit from these licenses, as they can become more popular and recognized by more people. By relinquishing certain controls (and possibly the revenues that go along with them) content creators can be invited to more shows, obtain more sponsorships, make more collaborations, etc.

Open content licenses can be practical in the internet age, because creators can publish their works without relying on another person or institution. For example, a photographer who wants to exhibit their photos could publish them on a personal website. This way, they can establish themselves and be known to a wider public without signing up an agency or gallery to do so.

By choosing the right license, rights holders can maximize distribution and maintain control of the commercialization of their works. If people want to use a work commercially, the content creator can preserve the right to grant or withhold permission. Even if others are prohibited from commercial use, rights holders may still use their work commercially.

Besides the possibility of a much broader distribution of a work, open content licenses also increase legal certainty for users and significantly decrease legal transaction costs.

People may also use funding models that do not depend on using a noncommercial license. For example, many artists and creators use crowdfunding to fund their work before releasing it under a permissive license. Others use a model where the basic content is free, but extras such as printed versions or special access to a members-only website are for paying customers only.

Open content licenses allow people to distribute their works to anybody and on any media and format such as websites, photocopies, CDs, or books, without restrictions.

Open content licenses automatically establish a license between authors and users. Without an open content license, the sharing of works via another online source would require an individual contractual agreement between authors and users.

Trademarks and Copyrights

Trademarks, like copyright, are a type of intellectual property. But the law that protects a trademark is different from that protecting copyright. Trademarks protect brands, related words such as brand names, logos, symbols, and even sounds and colors that are used to distinguish particular goods and services from others. Any special element used to promote and distinguish businesses and the services or products sold from others can be a trademark.

The holder of a trademark is generally allowed to prevent others from using these trademarked items if the public will be confused. Trademark law helps producers of goods and services protect their reputation, and protects the public by giving them a simple way to differentiate between similar products and services.

Trademarks can be registered officially or be automatically protected under common law. This means that a trademark exists as soon as it is being used, but a common-law trademark does not offer the same legal protection as a registered trademark. That’s why many companies register their trademarks.

Copyright and trademarks can coexist. For example, Wikipedia is a registered trademark, and its logo is also protected under copyright law as an original artwork or creation.

Guided Exercises

1. Can open content licenses be used to avoid copyright?

2. What is considered open content?

3. What is a derivative work?

Explorational Exercises

1. What would you do if you want to publish a work and allow people to use it for any purposes as long as they redistribute the work under the same rights and conditions?

2. Can you distribute commercial derivative works based on works published under an open content license?

Summary

In this lesson, you learned:

- Fundamentals of copyright law
- What can be considered copyrightable content
- What a derivate work or adaptation is
- Common features shared by open content licenses
- Types of open content
- Importance and benefits of the open content licensing model
- Copyright and trademarks as intellectual property types

Answers to Guided Exercises

1. Can open content licenses be used to avoid copyright?

   Open content licenses cannot be used to avoid copyright. Open content licenses are a type of copyright license that grant some rights under certain conditions, while maintaining the exclusive rights of the copyright holder.

2. What is considered open content?

   Open content can be any copyrightable work published under an open content license. This content can be movies, music, images, texts, databases, documentation, maps, and hardware designs, and other creations. These types of licenses grant rights such as use, redistribution, making derivative works, and commercial or noncommercial use without any special permission.

3. What is a derivative work?

   A derivative work, also called an adaptation, is a work based on an existing work in which the original work is transformed or adapted. Translations, musical arrangements, dramatizations, motion picture versions, sound recordings, and art reproductions are examples of derivative works.

Answers to Explorational Exercises

1. What would you do if you want to publish a work and allow people to use it for any purposes as long as they redistribute the work under the same rights and conditions?

   You can make the work available under a copyleft-type license. This way, all derivate works of the original must be open as well, requiring them to be published under the same or any other compatible license.

2. Can you distribute commercial derivative works based on works published under an open content license?

   It depends on the license under which the original work is available. If the license states that you can use derivative works for any purpose, even commercially, then you can.

053.2 Creative Commons Licenses

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 053.2

Weight

2

Key knowledge areas

- Understanding the concept of Creative Commons licenses
- Understand the Creative Commons license types and their combinations
- Understanding the rights granted by Creative Commons licenses
- Understanding the obligations created by Creative Commons licenses

Partial list of the used files, terms and utilities

- Public Domain Dedication (CC0)
- Creative Commons Attribution (CC BY)
- Creative Commons Attribution-ShareAlike (CC BY-SA)
- Creative Commons Attribution-NonCommercial (CC BY-NC)
- Creative Commons Attribution-NonCommercial-ShareAlike (CC BY-NC-SA)
- Creative Commons Attribution-NoDerivatives (CC BY-ND)
- Creative Commons Attribution-NonCommercial-NoDerivatives (CC BY-NC-ND)

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	053 Open Content Licenses
Objective:	053.2 Creative Commons Licenses
Lesson:	1 of 1

Introduction

The success of free software since the 1990s, along with the simultaneous triumph of the internet, have awakened a desire in other sectors for similar conditions to support the development and distribution of creative works, i.e. works that are fundamentally subject to copyright. These desires reflect a variety of interests.

Creative artists want to determine for themselves the conditions under which their works can be used by others. At the same time, however, they also have an interest in using the work of others for themselves, as creative processes have always done, for example by quoting, editing, or adapting. For the recipients of the works, questions arise as to the possibilities for use, for example whether and in what form a work may be reproduced or passed on.

In addition to clarifying such practical questions, regulations must be made in such a way that they are in line with applicable legal norms — primarily copyright law — which is made more difficult by the diverse ways copyright law is regulated internationally.

Last but not least, the rules should be easy to understand and simple to apply, in order to speed up creative processes and give creators and recipients legal certainty.

Origin and Goals of Creative Commons

The early licenses in the area of free software, above all the GNU General Public License, had done pioneering work, as they had succeeded in providing a reliable legal framework for the completely new processes surrounding the collaborative development of software.

In 2001, a group led by law professor Lawrence Lessig from Stanford Law School founded a non-profit organization called *Creative Commons* (CC), inspired by free software. The organization's goal is summarized on the project website as follows:

> Creative Commons is a global nonprofit organization that enables sharing and reuse of creativity and knowledge through the provision of free legal tools.
>
> — Creative Commons, Website (FAQ)

With the term "commons," referring to a form of communal economic activity already documented in the Middle Ages, the organization emphasizes a historically verifiable, deeply human need for collective action that is oriented toward the common good. Their self-imposed task is therefore to create a modern legal framework for creative work. Creative Commons transfers the demands and solutions of the free software movement to other creative fields such as literature, the performing arts (painting, graphics, photography, video, etc.), and music, but also documentation and scientific work — roughly speaking, all works that are subject to copyright.

Similar to the Free Software Foundation, Creative Commons also chooses to implement its goals through *licenses*: compilations of standardized, legally binding specifications that creators explicitly assign to their works, usually before they "release" works to the public, i.e. publish them.

The "all rights reserved" principle enshrined in US copyright law is perceived as too restrictive in view of the constantly growing technical possibilities available to creative processes. Creators themselves are often unclear as to how far they are allowed to build on the work of others without exceeding the limits of copyright law and, in the worst case, making themselves liable to prosecution. Creative Commons contrasts this heavy-handed regime with the principle of "some rights reserved": The creators name those rights that they reserve — and thus waive all others.

The combination of just four simple basic conditions (modules) results in a total of six licenses from which authors can select and assign the appropriate one for their work.

The Creative Commons License Modules

The four modules just mentioned correspond to four simple basic decisions that an author makes with regard to the use and distribution of a work. Each module can be represented by two-letter abbreviation and a symbol. The respective definition is very short and understandable even for legal laypersons, which contributes greatly to the popularity of CC licenses. In individual cases, however, the clauses can lead to unresolved questions or gray areas.

Attribution (BY)

> You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
>
> — Creative Commons, Attribution

The English word “by” indicates that the authors must be named, i.e. the work must be clearly attributed to the authors if it is used, reproduced, or passed on in any form. The BY module is the only one that is mandatory in *all* CC licenses. In other words, there is no work under the six standard CC licenses that can escape attribution.

This simple requirement can cause practical problems in collaborative projects developed over the internet. For example, in works derived from the online encyclopedia Wikipedia, it is quite questionable how “attribution” is to be “appropriately” carried out with thousands of contributors.

NonCommercial (NC)

> You may not use the material for commercial purposes.
>
> — Creative Commons, NonCommercial

The intention of the *NonCommercial* module seems clear at first glance: It is intended to prevent a freely available work from being appropriated by others for commercial purposes. This often concerns works that have been developed with public funds, for example at universities. These are meant to be “protected” from commercialization in order to ensure their quality and long-term free availability.

In fact, the NC module often causes uncertainty in practice, as it is by no means clear in individual cases when a use is actually commercial. Teachers who use materials under a CC license with an NC module, for example, are already in a legal grey area if they teach at a private school or a private university that students have to pay to attend.

Many critics of the NC module also reject the argument that free materials become “non-free” through commercial use, because the works are still *also* available free of charge.

NoDerivs (ND)

> If you remix, transform, or build upon the material, you may not distribute the modified material.
>
> — Creative Commons, NoDerivs

Derivatives (short: *Derivs*), meaning derivative works, have already been mentioned in other lessons. In the context of Creative Commons, the term also refers to the creation of new works on the basis of existing works covered by copyright; these can be learning materials that a teacher adapts for his lessons, for example, or the translation of a novel into another language. *NoDerivs* categorically excludes the distribution of modifications or adaptations of this kind: The work may be passed on only in the form published by the author.

Share Alike (SA)

> If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
>
> — Creative Commons, ShareAlike

The *ShareAlike* module indicates that a work may be shared only under the same conditions as those of the assigned license. SA is regarded as the counterpart to the copyleft principle in the free software licenses, which ensures that the freedoms granted by the license also apply unchanged to derived works.

The SA module is particularly interesting in combination with other modules because the SA requirement also updates other defined conditions, as we will see as we discuss the various licenses.

The Creative Commons Core Licenses

The combinations of the modules just presented result in a total of six licenses. There are only six because not all possible combinations of the four modules make sense: For example, the requirements to share even modified works under the same conditions (ShareAlike) and the prohibition of adaptations (NoDerivs) are mutually exclusive. Consequently, there is no license that contains both modules. Since the attribution of the author is also a component of all licenses, the result is the so-called *core licenses* we'll discuss next.

The list of these licenses can also be imagined as a scale from “many freedoms” to “few freedoms,” since an author answers questions about the possibilities of use or their restrictions in relation to the work step by step and thus finally receives the desired license.

Creative Commons Attribution (CC BY)

The attribution of the author has already been mentioned as a mandatory component of all licenses, and so the first license, *CC Attribution*, includes only this module. A work is thus released for any form of use, modification, and distribution as long as the attribution requirement is met.

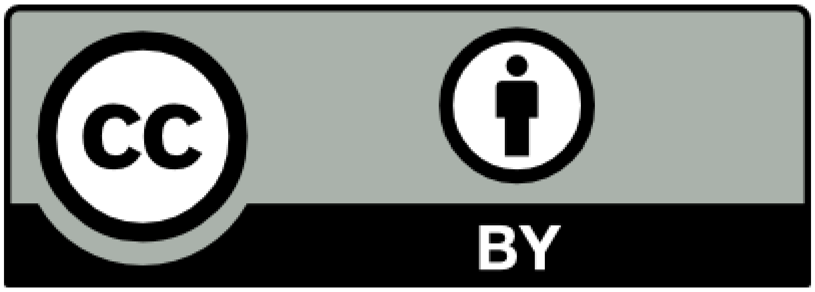


Figure 5. CC BY Icon

For example, if an author has published a poem under CC BY, a publisher could include it in a collection of poems without consulting the author and without any financial obligations and distribute it through bookstores, for example, as long as this poem is clearly marked as the author's work.

Creative Commons Attribution-ShareAlike (CC BY-SA)

CC Attribution-ShareAlike_ corresponds to CC BY, but supplements it with the requirements of *ShareAlike*, i.e. the distribution of modified versions of the work under the same conditions.

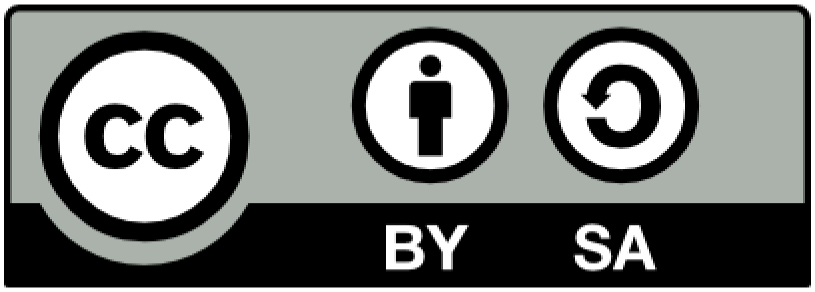


Figure 6. CC BY-SA Icon

For example, if a singer places her song under the CC BY-SA license, another band can cover or adapt this work, provided that they in turn publish their version of the song under under CC BY-SA or another license that is compatible with it.

Creative Commons Attribution-NonCommercial (CC BY-NC)

The *CC Attribution-NonCommercial* is the first license in the series that links the use of a work to a prohibition by excluding distribution and adaptation for commercial purposes.

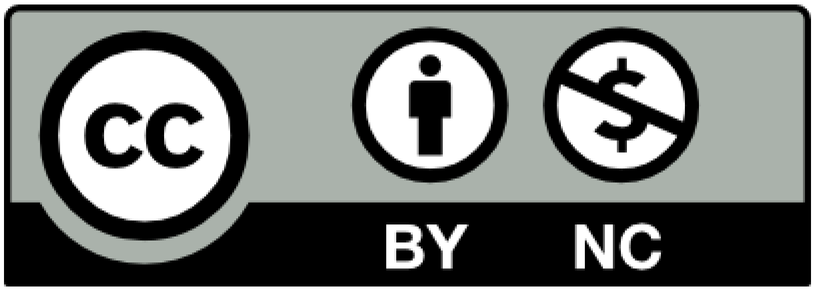


Figure 7. CC BY-NC Icon

In our example, a band would be allowed to adapt the singer’s song, but not to use this adaptation commercially, for example not to sell it on their own records or play it at concerts for which they receive a fee. However, the band would be allowed to additionally prohibit adaptations of their version, i.e. to publish the adaptation under the CC BY-NC-ND license.

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

In the case of *CC Attribution-NonCommercial-ShareAlike*, the prohibition of commercial use is linked to distribution under the same conditions. To stay with our example: Although the band is allowed to cover the singer’s song, it is not allowed to add ND to the cover version because the arrangement must be distributed under the same conditions.

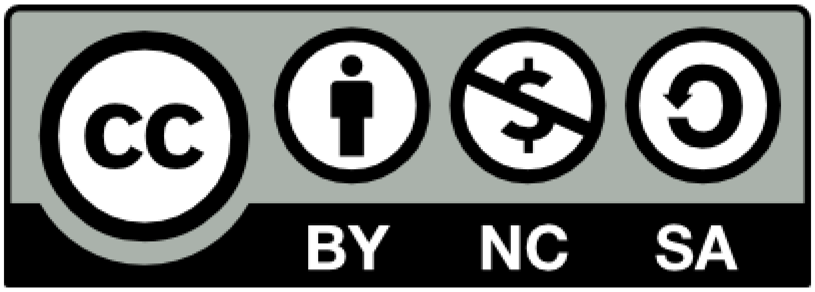


Figure 8. CC BY-NC-SA Icon

Creative Commons Attribution-NoDerivs (CC BY-ND)

Like NonCommercial, *CC Attribution-NoDerivs* is also based on a prohibition — in this case, the prohibition to modify or adapt a work. The band in our example would therefore not be allowed to distribute a cover version of the singer’s song.

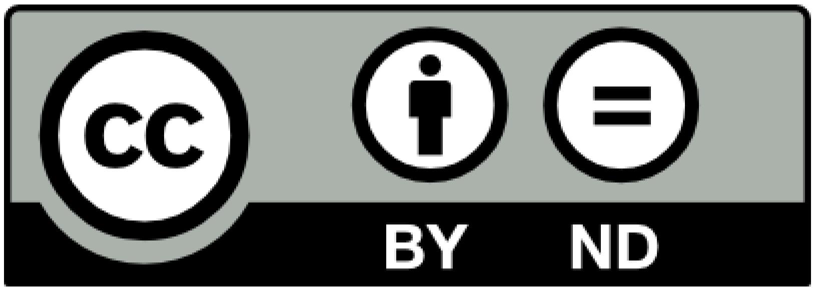


Figure 9. CC BY-ND Icon

The use of CC licenses with ND clauses (i.e. CC BY-ND license and the CC BY-NC-ND license presented next) is viewed with scepticism, particularly in the scientific field, because they can hinder the exchange of knowledge necessary for scientific progress. Even on the Creative Commons website, licenses with an ND module are described as incompatible with the principle of open access that is popular in science. As an example of the overly restrictive tendency of ND, it is cited that even translations of scientific articles are regarded as derivative works and are therefore prohibited.

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

The most restrictive of the CC licenses, *CC Attribution-NonCommercial-NoDerivs*, combines the prohibition of commercial use with that of the distribution of derived works. In positive terms, this means that a work may be reproduced and distributed only in the form published by the author and may be used only for non-commercial purposes.

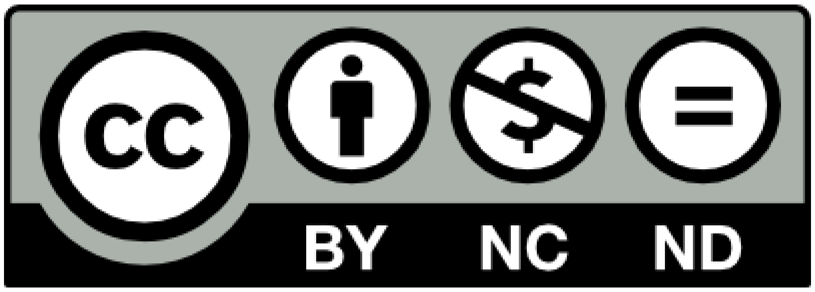


Figure 10. CC_BY-NC-ND_Icon

The singer in our example thus prevents a band from covering her song (ND), but at the same time risks her song being excluded from on any platform that generates revenue from customers or advertising.

Creative Commons Zero (CC0) and the Public Domain Mark

It has already been mentioned that copyright law is regulated very differently in different countries and jurisdictions. Under US law, for example, a rights holder can completely waive his copyright. By doing so, they transfer their work to the so-called *public domain*, i.e. to general and unrestricted use.

In most European countries, however, in addition to the pure rights of use, copyright also includes personal rights, which are generally not transferable and therefore cannot be waived by the author. If an author wishes to release a work to the public, they waive all rights of use, but remain the author.

In addition, there are works whose use is not subject to any restrictions per se. These include, for example, works whose legal term of protection has expired (such as a novel 70 years after the author's death), or works that were never protected in principle (such as legal texts).

Creative Commons provides two so-called *public domain tools* to identify works that are available to the general public without restriction.

Creative Commons Zero (CC0) attaches virtually zero conditions to the reproduction, distribution, and modification of a work. The obligatory attribution of the author in all CC licenses is also omitted. The license allows rights holders to mark their works as public domain in as many legal systems as possible.

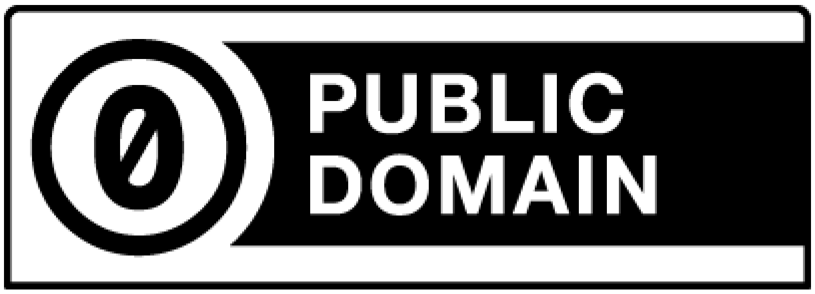


Figure 11. CC0 Icon

Let's take a learning document developed for teaching purposes as an example. The author or authors can use the CC0 label to ensure that this material can be used by anyone without any restrictions: It can be copied in whole or in part, distributed free of charge or for a fee, and integrated in whole or in part into other documents. The original authors do not have to be named anywhere.

To identify works that are not subject to copyright restrictions, Creative Commons recommends the *Public Domain Mark*:

> The Public Domain Mark operates as a tag or a label, allowing institutions like those as well as others with such knowledge to communicate that a work is no longer restricted by copyright and can be freely used by others.
>
> — Creative Commons, Public Domain Mark

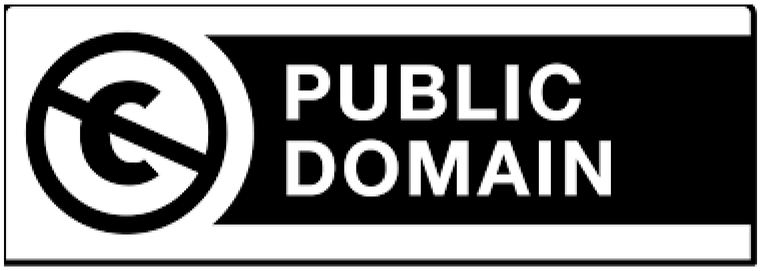
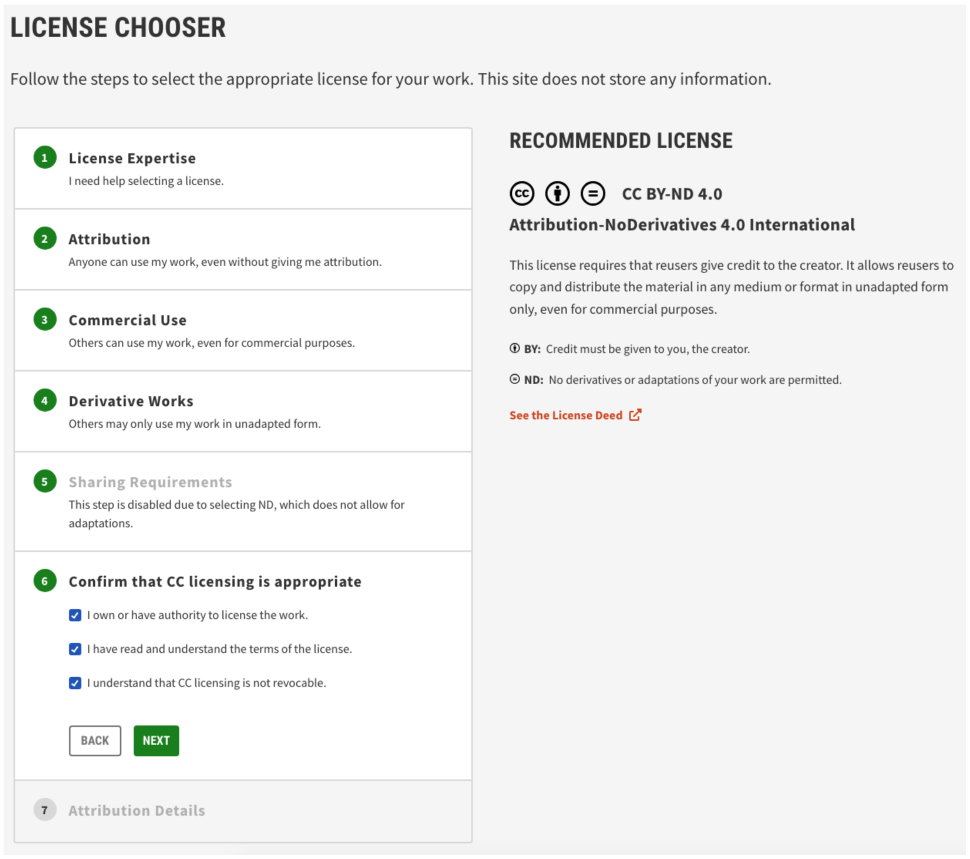


Figure 12. Public Domain Mark Icon

Selecting Licenses and Marking Works

It has already been mentioned that Creative Commons aims to be as simple as possible for both licensors (content creators) and licensees (recipients of the works). In concrete terms, the organization offers a great deal of assistance from the selection of the appropriate license to the labeling of the work and its use.

The *CC License Chooser* on the organization’s website leads step by step to the recommendation of the appropriate license by posing simple questions that an author answers in relation to the desired use of their work (CC License Chooser).



LICENSE CHOOSER

Follow the steps to select the appropriate license for your work. This site does not store any information.

1 License Expertise
I need help selecting a license.

2 Attribution
Anyone can use my work, even without giving me attribution.

3 Commercial Use
Others can use my work, even for commercial purposes.

4 Derivative Works
Others may only use my work in unadapted form.

5 Sharing Requirements
This step is disabled due to selecting ND, which does not allow for adaptations.

6 Confirm that CC licensing is appropriate
- [x] I own or have authority to license the work.
- [x] I have read and understand the terms of the license.
- [x] I understand that CC licensing is not revocable.

BACK NEXT

7 Attribution Details

RECOMMENDED LICENSE

CC BY-ND 4.0
Attribution-NoDerivatives 4.0 International

This license requires that reusers give credit to the creator. It allows reusers to copy and distribute the material in any medium or format in unadapted form only, even for commercial purposes.

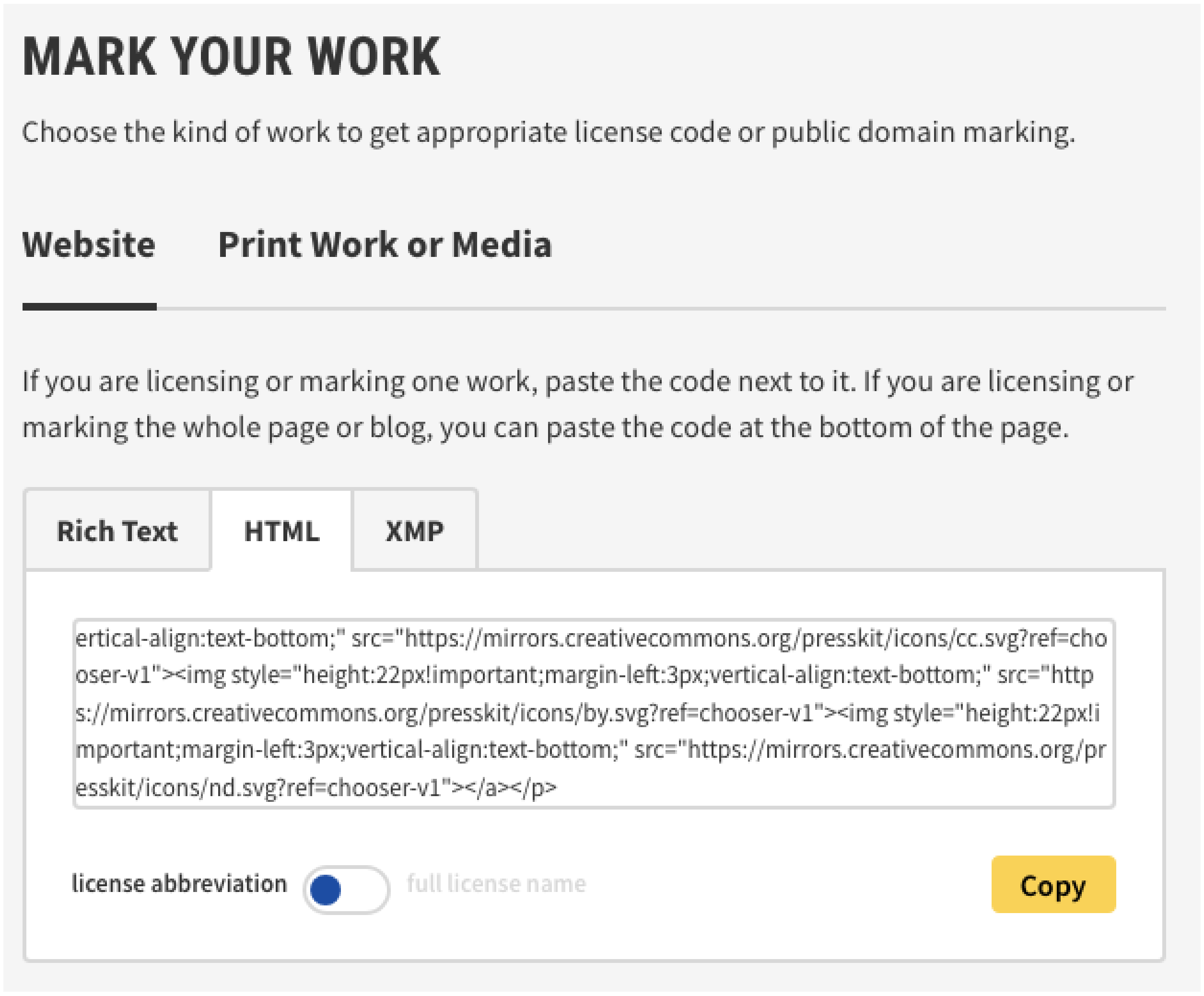
BY: Credit must be given to you, the creator.

ND: No derivatives or adaptations of your work are permitted.

See the License Deed

Figure 13. CC License Chooser

If the proposed license corresponds to the author’s intentions, they can mark their work accordingly. The License Chooser provides HTML code, for example, which the author can add directly to a website displaying the work (HTML code with link to the license).



MARK YOUR WORK

Choose the kind of work to get appropriate license code or public domain marking.

Website Print Work or Media

If you are licensing or marking one work, paste the code next to it. If you are licensing or marking the whole page or blog, you can paste the code at the bottom of the page.

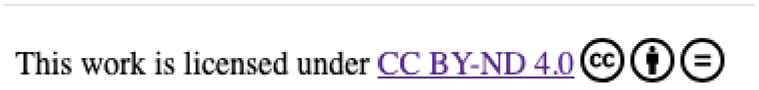
Rich Text HTML XMP

```
ertical-align:text-bottom;" src="https://mirrors.creativecommons.org/presskit/icons/cc.svg?ref=cho
oser-v1"><img style="height:22px!important;margin-left:3px;vertical-align:text-bottom;" src="http
s://mirrors.creativecommons.org/presskit/icons/by.svg?ref=chooser-v1"><img style="height:22px!i
mportant;margin-left:3px;vertical-align:text-bottom;" src="https://mirrors.creativecommons.org/pr
esskit/icons/nd.svg?ref=chooser-v1"></a></p>
```

license abbreviation full license name Copy

Figure 14. HTML code with link to the license

The result is then a standardized notice with the link to the corresponding license (License notice with link to the license).



This work is licensed under CC BY-ND 4.0

Figure 15. License notice with link to the license

The license-specific icons, in which the included modules are directly visible, have already been introduced. They have established themselves as “eye-catchers” identifying free content on the internet.

International and Ported Licenses

From the outset, Creative Commons has focused on the development of licenses that are as general as possible, i.e. valid worldwide, which it identifies accordingly with the addition of “international” (formerly also “unported”). On the other hand, there are variants that take into account the peculiarities of a regional or national legal system and are referred to as “ported.” For example, in addition to the *CC BY-SA 3.0 Unported* license, there is also a special version for Germany under the name *CC BY-SA 3.0 Germany*.

With the currently valid version 4.0 of the CC licenses, Creative Commons is striving for further standardization and has (so far) completely dispensed with ported versions. In version 4.0, the “international” license: is expressly recommended:

> We recommend that you use a version 4.0 international license. This is the most up-to-date version of our licenses, drafted after broad consultation with our global network of affiliates, and it has been written to be internationally valid. There are currently no ports of 4.0, and it is planned that few, if any, will be created.
>
> — Creative Commons, FAQ

Guided Exercises

1. Which module of the Creative Commons license system is part of all six Creative Commons licenses?

2. Which module of the Creative Commons license system requires the distribution of a work under the same conditions?

3. How does CC0 differ from the six CC core licenses?

4. What is the difference between “international” and “ported” licenses?

Explorational Exercises

1. Is it possible to place a photograph of a work that is in the public domain under a CC license?

2. Can an author publish his novel, in which he incorporates a Shakespearean sonnet in its entirety, under a CC license?

3. A user publishes a photo of herself on her website under a CC license. Can she prevent the distribution of this image by invoking her personal rights?

Summary

Founded in 2001, the Creative Commons organization aims to support the sharing and use of creative works, i.e. works subject to copyright, with the help of free legal tools. The focus is on four modules (Attribution, NonCommercial, NoDerivs and ShareAlike), which are combined in six so-called core licenses and can be selected by authors for their works.

Answers to Guided Exercises

1. Which module of the Creative Commons license system is part of all six Creative Commons licenses?

   The “Attribution” module, abbreviated “BY.”

2. Which module of the Creative Commons license system requires the distribution of a work under the same conditions?

   The “Share Alike” module, abbreviated to “SA.”

3. How does CC0 differ from the six CC core licenses?

   With CCO, an author does not secure certain rights — as with the other CC licenses — but waives *all* rights to the work within the scope of the possibilities of the respective jurisdiction. This principle of “no rights reserved” corresponds to the designation of the work as *public domain*.

4. What is the difference between “international” and “ported” licenses?

   Until version 3.0, Creative Commons offered variants of its licenses that took into account the specifics of a regional or national jurisdiction. Since 4.0, CC has dispensed with these so-called “ported” versions and recommends the globally valid “international” version of the respective license.

Answers to Explorational Exercises

1. Is it possible to place a photograph of a work that is in the public domain under a CC license?

   This depends on whether the photograph fulfills criteria according to which it is itself protected by copyright. In other words, the photograph itself must be recognizable as a creative work and worthy of protection, for example through perspective, background, filters, etc. If this is the case, it can be placed under a CC license and the conditions of the CC license then apply. However, the original part, i.e. the actual motif, remains in the public domain.

2. Can an author publish his novel, in which he incorporates a Shakespearean sonnet in its entirety, under a CC license?

   Yes, but only the author's own creative parts of the novel are subject to the CC license chosen by the author. The sonnet, which is in the public domain, remains in the public domain.

3. A user publishes a photo of herself on her website under a CC license. Can she prevent the distribution of this image by invoking her personal rights?

   By publishing a photo under a CC license, an owner generally agrees to its distribution. A limit is reached when the use of the photo violates the personal rights of the owner, for example when the image is grossly distorted or used for advertising or in a political context without the owner's consent. The personal rights of the owner are not affected by the regulation of the rights of use by the CC license, so that in such cases she can take legal action against the use of her photo.

053.3 Other Open Content Licenses

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 053.3

Weight

1

Key knowledge areas

- Understanding licensing for documentation
- Understanding licensing for data sets and databases
- Understanding the rights granted by open content licenses
- Understanding the obligations created by open content licenses

Partial list of the used files, terms and utilities

- GNU Free Documentation License, version 1.3 (GFDL)
- Open Data Commons Open Database License (ODbL)
- Community Data License Agreement – Permissive, version 1.0 (CDLA)
- Community Data License Agreement – Sharing, version 1.0 (CDLA)
- Open Access

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	053 Open Content Licenses
Objective:	053.3 Other Open Content Licenses
Lesson:	1 of 1

Introduction

Previous lessons have already introduced the concept of open content, focusing on Creative Commons licenses. In addition to Creative Commons licenses, however, there are other open content licenses, some of which relate to specific content and are also regularly used for “by-products” (such as documentation) of open source projects.

Licenses for (Software) Documentation

Large software repositories regularly contain documentation or manuals for the software in addition to the code. Documentation usually contains various copyrightable elements, such as explanatory texts, illustrative graphics, or code examples. Open source licenses that apply to the contents of a repository are generally also applicable to such texts if no other information is available; for example, GPLv3.0 applies to “any copyrightable work” and is not limited to code.

However, documentation is often subject to different license conditions. This can make sense for various reasons, because documentation is produced and used differently from code. The license conditions of a strict copyleft license for computer programs may, for instance, prevent the adoption of individual parts of the documentation for other programs. It is also unclear what is meant when the “source code” is to be provided for GPLv3.0-licensed documentation. In addition, it used to be common practice to distribute printed versions of documentation or manuals for programs. This circumstance can be better taken into account by specific documentation licenses (for example, if some license obligations are linked to a certain number of distributed copies).

Creative Commons licenses are often used for documentation, but there are also special copyleft licenses for documentation, such as the *Free Documentation License* (FDL), which explains its purpose in its preamble:

> The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.
>
> This License is a kind of “copyleft,” which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.
>
> We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.
>
> — Free Documentation License, Preamble

The license is very specific to documentation; for example, it also addresses printed versions. In particular, a copyleft documentation license explicitly does not affect the code with which it is delivered. Although it would also be possible to distribute documentation under GPLv3.0 together with MIT-licensed code: Since in this case there would be two independent works, the GPLv3.0 of the documentation would not result in the code having to be licensed under GPLv3.0. But a dedicated license for documentation provides more clarity in this regard.

Licenses for Databases

There are also special licenses for databases. Among other things, these licenses take into account the use of a database as a resource for a program and the special features of copyright protection for databases.

What the Term “Database” Applies to

In copyright law, a distinction is made in some legal systems (including in Europe) between database works that, like other copyright-protected works (images, texts, etc.), enjoy protection on the basis of *personal intellectual creation* (selection and arrangement of elements), and databases for whose production (procurement, verification, presentation) a *substantial investment* was required (database right).

In both cases, the object of protection is a collection of independent elements (such as distinct works or items of data) that are systematically or methodically arranged and individually accessible electronically or by other means. The copyrightability of the elements themselves is not important, so they can also be mere numerical values.

A collection of data can therefore be protected by copyright as a database work or under database right (in the EU). For a database work to be protected, the arrangement and selection of the elements must express a personal intellectual creation. This is unlikely to be the case for data collections that are used as the basis for software — but these can be protected under database right. To earn this right, a substantial investment must have been made in the collection and arrangement of the elements. This investment can also consist of time and human resources.

Various licenses are now available for databases and data collections that better cover the use of data than “classic” open source software licenses. Let’s take a look at the distinction using an example.

As part of an academic research project, all of an author’s poems are to be collected and published on a website, where they are sorted alphabetically by title and can be accessed individually. However, the project requires that the relevant copyrights for the distribution and reproduction of the poems are obtained from the author. License fees are paid for this.

The license fees may constitute a “substantial investment.” Since the poems are individually retrievable and arranged systematically (according to the alphabet), the database serving the poems is protectable under database right. However, the elements (poems) were not specifically selected (the selection was done on the basis of completeness) and the arrangement is not creative, but purely systematic. There is therefore no personal intellectual creation in relation to the database, so that the database does not qualify as a database work.

If, on the other hand, the poems had been arranged and selected according to certain criteria — for example to document certain motifs in the author’s work in the context of the creative period — a personal intellectual creation could exist, and then this collection of poems would be protectable as a database work.

However, the two categories are treated differently under copyright law, in particular with regard to the term of protection: databases under the database right are protected only 15 years after production/publication, as opposed to 70 years after the death of the author for database works.

Delimitation for Data

The *data* contained (in our example, the poems) must always be considered separately from the database as a whole, because they may (but need not) be eligible for protection in their own right and may therefore be subject to different license conditions. For example, the collection of poems may be subject to the Open Database License (see below), while the individual poems may be subject to a Creative Commons license or a proprietary license.

Delimitation for Database Management System (DBMS)

In addition, the software for managing the data, the so-called *database management system* (DBMS), which is not considered part of the database under copyright law, but as a means of accessing the data, must be considered separately under copyright law. Such applications — such as MySQL, PostgreSQL, MariaDB, and others — are to be considered separately as computer programs and are protectable as such.

Datasets for Training Machine Learning Models

Various collections of data can be downloaded from websites such as huggingface.co or kaggle.com, for example to train machine learning models. Such collections, also known as *datasets*, are usually subject to database right, provided a “substantial investment” has been made in their production (which is usually not readily apparent from the outside).

The term “dataset” is somewhat vague because it can refer to a line in a database — i.e., an entry or a definable element — but also to collections of data or datasets.

Open Database License (ODbL)

These distinctions are also visible in the *Open Database License* (ODbL), a widely used license in the database sector. The ODbL was developed by Open Data Commons, a project of the Open Knowledge Foundation. The license makes a clear distinction in its preamble between protection of the database and protection of the individual contents of the database. The license refers only to the former, while the contents can also be licensed elsewhere independently of this.

Of course, it is still possible to choose the same license for the database and the content, e.g., to compile a collection of works licensed under CC-BY-4.0 in a database that is in turn licensed under CC-BY-4.0.

The best license for the database depends on the objectives of the database producer. To stay with our example of the poetry collection: If the poetry database is to be modified at will and may also be distributed under other license conditions, the ODbL would not be the right choice. This is because the ODbL contains a copyleft for databases, which means that a database derived from an ODbL-licensed database (e.g. a database that adopts all elements and the arrangement of the elements of the original database and adds others) may be redistributed only under the same conditions. In this case, a Creative Commons license with editing rights (e.g. CC-BY) is a better choice.

The ODbL also makes it clear that it in no way applies to computer programs that manage the database (i.e., a DBMS): “This License does not apply to computer programs used in the making or operation of the Database.”

However, the license stipulates that a work created from the contents of the database and available for public use (a so-called *produced work*) must at least indicate which database (not DBMS) was used for it and that the the database itself may be used under the conditions of the ODbL. An example of such a *produced work* is a street map created from the coordinates collected in the database.

If one of the known (software-specific) copyleft licenses such as GPLv3.0 were used for a database, it can be assumed that a computer program that uses the database (e.g. for a store system) could be distributed only under the terms of GPLv3.0. The license of a DBMS, on the other hand, is likely to be independent of the license of the database because the DBMS is not considered a “derived work” of the database; it merely enables access and editing and is therefore an independent work.

Community Data License Agreements (CDLA)

The *Community Data License Agreements* (CDLA), a project of the Linux Foundation, also focuses on the licensing of data (as distinct from software). Training data collections for machine learning systems can also be the subject of CDLAs.

> Our communities wanted to develop data license agreements that could enable sharing of data similar to what we have with open source software. The result is a large scale collaboration on licenses for sharing data under a legal framework which we call the Community Data License Agreement (CDLA).
>
> These licenses establish the framework for collaborative sharing of data that we have seen proven to work in open source software communities.
>
> — CDLA website (cdla.dev)

The CDLA is therefore based on a framework that takes into account different levels of licensing issues, especially when data is compiled from multiple sources by different contributors. In our example, this could be a collection in which different authors contribute their own poems to build a common poetry database.

CDLA distinguishes between the “inbound license” for data that is added to the collection and the “outbound license” for the use of the data. At a third level, it specifies requirements for the organization that hosts or holds the data.

To stay with the poetry example: An author who contributes their poem would be guided by the specifications for “inbound licenses”. The “outbound license” concerns the conditions agreed upon by the editors of the poetry collection as a whole.

Similar to the open source licenses for software, CDLAs are available in the *permissive* and *sharing* categories, which we will briefly introduce. A visit to the aforementioned kaggle.com or huggingface.co websites will give you an impression of which licenses are used for databases or datasets: For example, you can use filters to find out which data collections are offered under CDLA.

Community Data License Agreement — Permissive

Version 2.0 of the permissive CDLA has been available since 2021 and is much more concise and clearly formulated than version 1.0.

Version 1.0 grants rights to use and publish the licensed data, including the publication of so-called “enhanced data,” which also includes edits or additions to the dataset. When passing on or publishing the data, basic license obligations must be observed; among other things, the license text must be passed on, changes must be marked, and copyright notices must be retained. The data can also be passed on under additional or other license conditions.

In version 2.0, the wording “enhanced data” is no longer used; instead, the granting of rights is formulated more precisely:

> A Data Recipient may use, modify, and share the Data made available by Data Provider(s) under this agreement if that Data Recipient follows the terms of this agreement.
>
> — Community Data License Agreement - Permissive, version 2.0

The only condition for sharing the data in version 2.0 is the passing on of the license text or a link to the license text.

Community Data License Agreement — Sharing

The sharing version of the CDLA, which is currently only available in version 1.0, contains a copyleft for data in such a way that the transfer (“publish”) of “enhanced data” is permitted only under the same conditions.

> The Data (including the Enhanced Data) must be Published under this Agreement in accordance with this Section 3;
>
> — Community Data License Agreement -- Sharing, version 1.0

Open Access

One term that is often used in connection with open content is *open access*. Although the term is not defined in the legal sense, the declaration of the Budapest Open Access Initiative (BOAI) makes the intention behind open access clear. The initiative arose from a conference in Budapest in 2001 and summarizes the international efforts for open access:

> The literature that should be freely accessible online is that which scholars give to the world without expectation of payment. Primarily, this category encompasses their peer-reviewed journal articles, but it also includes any unreviewed preprints that they might wish to put online for comment or to alert colleagues to important research findings. There are many degrees and kinds of wider and easier access to this literature. By “open access” to this literature, we mean its free availability on the public internet, permitting any users to read, download, copy, distribute, print, search, or link to the full texts of these articles, crawl them for indexing, pass them as data to software, or use them for any other lawful purpose, without financial, legal, or technical barriers other than those inseparable from gaining access to the internet itself. The only constraint on reproduction and distribution, and the only role for copyright in this domain, should be to give authors control over the integrity of their work and the right to be properly acknowledged and cited.
>
> — Declaration of the Budapest Open Access Initiative

Open access therefore refers in particular to free access to scientific literature and other content on the internet. The open access movement originated in the 1990s with the aim of making scientific publications available to the general public. Open access therefore does not refer to a specific license or certain licensing conditions, but rather to a concept of licensing.

When an article is published with open access, the first step is to choose an open content license, such as one of the Creative Commons licenses. However, the choice of a CC license is not mandatory for open access. Over the years, a categorization of different open access strategies has developed, which also take dual licensing into account — for example, granting a comprehensive license to a publisher while simultaneously the author offers the content for download on their own website.

Both the BOAI and the *Berlin Declaration* that followed two years later are important milestones in the open access movement. The preamble to the Berlin Declaration states:

> In accordance with the spirit of the Declaration of the Budapest Open Access Initiative, the ECHO Charter and the Bethesda Statement on Open Access Publishing, we have drafted the Berlin Declaration to promote the Internet as a functional instrument for a global scientific knowledge base and human reflection and to specify measures which research policy makers, research institutions, funding agencies, libraries, archives and museums need to consider.
>
> — Berlin Decalaration 2003

Guided Exercises

1. Can “classic” open source licenses also be used for documentation and databases?

Yes	
No	
It depends, there is no clear answer	

2. Does it make sense to use separate licenses for documentation? Why?

3. Does it make sense to use separate licenses for databases? Why?

4. Which of the following licenses are suitable for documentation?

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

5. What is meant by the term “open access”?

Explorational Exercises

1. Are there specific licenses for fonts?

2. Briefly explain two open access strategies or models.

3. What is meant by the “open definition”?

Summary

In addition to the well-known Creative Commons licenses, there are other concepts for licensing content that are even more specifically tailored to the respective data types: For software documentation or databases, the FDL or ODbL licenses are available that take into account the characteristics of these types of works. This lesson provides an overview of some of these licenses and also addresses the concept of open access.

Answers to Guided Exercises

1. Can “classic” open source licenses also be used for documentation and databases?

Yes	X
No	
It depends, there is no clear answer	

2. Does it make sense to use separate licenses for documentation? Why?

   Yes, because they usually contain other content than the program itself. They also regulate how printouts of the documentation are to be handled, for example.

3. Does it make sense to use separate licenses for databases? Why?

   Yes, it can be useful. In particular, a specific database license can achieve a copyleft effect for the database without the copyleft affecting the surrounding code. In addition, database licenses address specific provisions that copyright law provides for the protection of databases.

4. Which of the following licenses are suitable for documentation?

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

5. What is meant by the term “open access”?

   The term describes free access to scientific literature and other content on the internet.

Answers to Explorational Exercises

1. Are there specific licenses for fonts?

   Yes, there is, for example, the SIL Open Font License (OFL), which contains a copyleft specifically for fonts, but this has no effect if fonts are used unchanged.

2. Briefly explain two open access strategies or models.

   In the “Gold” open access strategy, the publication is made available directly by the publisher under an open content license, usually (but not exclusively) using Creative Commons licenses.

   The “Green” open access model applies when the publisher does not distribute the publication under open access, but the author is given the opportunity to make the publication available for download, such as on their website under an open license.

3. What is meant by the “open definition”?

   The “open definition” is a common understanding, formulated by the Open Knowledge Foundation, of the term “open” in “open data,” “open knowledge,” and “open content.” The definition describes in particular the basic freedoms that must be granted if a license is to be “open” in the sense of the definition. These include access, use, editing or modification, and sharing.

Topic 054: Open Source Business Models

054.1 Software Development Business Models

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 054.1

Weight

2

Key knowledge areas

- Understanding goals and reasons to release software or content under an open license
- Understanding common business models and revenue streams for organizations developing open source software and open content
- Understanding the implications of using open source software as components in larger technology products and services
- Understanding the impact licenses have on a software development business models
- Awareness of cost structures and investments needed for open source software development business models

Partial list of the used files, terms and utilities

- Paid development
- Open-core and paid add-ons
- Freemium
- Enterprise and community versions
- Self-hosted distribution
- Subscriptions
- Customer support

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	054 Open Source Business Models
Objective:	054.1 Software Development Business Models
Lesson:	1 of 1

Introduction

Traditionally, companies that distributed proprietary software (also known as *closed source* software) had a relatively straightforward business model: They sold a license to use the software, either as a one-time fee or on an ongoing basis in a *subscription* model.

But over the decades, the market for software has changed radically, and customers today expect to pay a few dollars for an app and receive free updates for life.

As a result, the old-school approach to proprietary software isn't profitable anymore, and even companies that used to build their entire business model around selling software licenses have diversified into vendor integration, support, services, Software-as-a-Service (SaaS), cloud/container hosting, and hardware products that run software (phones, laptops, servers, printers, cars, smart watches, etc.).

In contrast, free and open source software licenses give anyone the right to use, study, modify, and redistribute the software without paying a fee. So, businesses based on open source never had the option to sell a license to use the software. You shouldn't expect to simply take the work of others from an open source project and make a huge profit giving that publicly available work to your customers, because they can get the same software for free directly from the project. Instead, companies that provide free and open source software use alternative business models. The most successful software business models often combine multiple forms of customer value.

This lesson looks at why a company would choose some of these software business models with open source software, and the trade-offs in these options.

Goals and Reasons to Release Software or Content Under an Open License

There are many reasons developers choose to build a business on open source code: The software may meet a need, solve a problem, or provide some features that the developers deliver to customers. A company can save time and money by sharing the work on software with other developers, instead of writing the same software from scratch. Participating in the project is an investment that hopefully pays off in usable, stable, reliable, secure, well-maintained software.

Many open source developers hope to get bug fixes from their customers. Some customers take their contributions to an even higher level by adding new features, porting the software to a new environment, or making other significant improvements. Both bug fixes and higher-level improvements enhance the original software. If customers contribute them back to the original developers, the changes might make the code more desirable to other potential customers and encourage its adoption.

But a company can’t expect contributions and attention just by placing its code on a site such as GitHub or GitLab. It’s one thing to encourage developers to join a project, but more challenging to keep them enthusiastic about it — so don’t underestimate the amount of work that community contributions call for. A company needs to put effort into recruiting developers, mentoring them, motivating them, and checking their code for flaws.

Another reason to open the code is to create an industry standard. Sharing code in an open manner could lead to better interoperability and other benefits. The team that developed the code has valuable expertise that might allow them to direct the future of the project and get paid for supporting others who use the code.

Some companies open their code in order to create trust among customers. First of all, customers know they can continue to use and improve the code if the company fails or decides to enter a different market and abandons the code. Second, customers can check the quality and security of the code for themselves or ask an independent expert to do an audit, which is usually not possible with closed, proprietary software.

Finally, a company might have adopted a code base that is already open source, planning to build a commercial business on it. The license might require the company to distribute the source code of its changes to any end-user of the program.

In short, some reasons to open a company’s code are:

- Building on an existing free software project
- Benefiting from customer innovation and contributions
- Creating trust among customers
- Promoting code as an industry standard

Common Business Models and Revenue Streams

The software industry, over the past several decades, has discovered many business models appropriate for proprietary as well as open source businesses. This section focuses on the most popular ones in current use among open source developers.

One business model, common in both proprietary and open source software, is to charge for support. Customers might have trouble installing and configuring the software, fixing bugs as they are found, adding new features for their exclusive use, and managing their systems. The staff who developed the software are excellent sources of support. In fact, this staff is probably already offering free support on forums where the software is discussed.

Thus, in the support model, a customer pays a vendor to install the open source software on the customer’s hardware (called a *self-hosting distribution*) or gets access to the software on the vendor’s hardware (a cloud-based model). Support contracts ensure that customers will get timely help from the company for their more complex needs. Companies using this model generally offer retainers, which clients pay for on a regular basis.

Another business model for proprietary software — really a set of overlapping models — is called *freemium*, a term popularized in 2009 by author and editor Chris Anderson, and based on a much older business model known as *razor and blades*. In the razor and blades model, customers pay very little for the base product (for example, a razor or a printer), and then pay a premium for necessary components designed to wear out (blades for the razor, ink cartridges for the printer).

In the freemium model, customers can use a product at a certain level cost-free, and are encouraged to pay for more features if they find the product useful. You probably know online news sites that offer a certain number of free articles and ask readers to subscribe for full access to their sites; this is a well-known example of a freemium model. Another example is a gaming platform that provides the base game for free, but charges money for specific inventory.

Other proprietary software companies base their freemium model on time: Try out the software free for three months and then pay in order to continue using it.

Proprietary software companies sometimes use a version of the freemium model known as *open core*. In open core, certain basic features (the “core” of the product) are open source, and additional proprietary features can be licensed.

Often, the open part is fully functional but works best for researchers or individual users, and becomes hard to manage when many people are sharing it across an enterprise. Therefore, the proprietary features might include convenient web interfaces for administration, tools for accounting and collaboration, and other things of particular interest to large sites.

Even though open core business models use some open source software, customers are smart enough to recognise that the whole product is actually proprietary. So, while open core may have the advantage of building on existing open source project, it fails to provide any other advantages of open source software. An open core business model doesn’t build trust with customers, inspire them to contribute to the base software, give customers access to examine the code, build a network of skilled developers outside the main company, or work as an industry standard. Even worse, the open core development model has the same costs of developing, without the benefits that make it worthwhile. Companies who try an open core business model are generally disappointed in the results, and many later switch to a pure proprietary model.

Companies can also build a web service on top of an open source or proprietary code base, in the *Software as a Service* (SaaS) model. Customers pay on a monthly or yearly basis.

Many companies who run SaaS web services or offer mobile apps make money through advertising. Some companies also collect data on users through the service or app and sell this data to third parties who can use it for advertising. Advertisements can be seen as annoying, though. Selling data is controversial, and some countries put restrictions on data collection.

Finally, companies might develop and release open source software without trying to derive any revenue from it at all. This model is available to companies who get revenue from things besides the software. For instance, automobile companies collaborate to create large bodies of free software to run in their cars (which nowadays are thoroughly computerized).

Major software companies who derive their revenues from proprietary offerings, such as Google and Amazon, sometimes release administrative software or other useful tools as open source, because these open source tools are not central to their core business. The organizations who release the code under an open license benefit from feedback, bug reports, and feature enhancements provided by the open source community.

Using Open Source Software in Other Technologies and Services

A lot of companies incorporate open source software into their products or web platforms. After all, open source software is free! But that’s not the most important reason (and perhaps not even a good reason) to adopt it. More importantly, a lot of this software is high-quality (although the development team should vet it carefully before adopting it) and may even be an industry standard.

Another advantage of open source software is that it will continue to be available if the developers or the community around it go away.

But free and open source software brings responsibilities. This section will quickly list the main issues to consider before adopting it.

The first issue applies to any third-party software or tool under consideration for adoption: Will it meet the users' needs now and as the company evolves? Is the software supported by a strong community that can offer technical support and continue to develop the software? Does the software suffer from significant security vulnerabilities?

Next, managers must consider the company’s responsibilities if it incorporates an open source project into its own software. If developers build new code around the open source code, they should check what they are required to do by the license from the open source software. Some licenses require developers to distributed the source code of their own changes to their end-users under the same free license as the original code base.

Even if a developer isn’t required to distribute their modified source code, and is creating a proprietary product on top of the open source code, the developer might want to contribute certain things back, such as bug fixes and enhancements to the open source code. By contributing these fixes and enhancements, the contributing developer allows the project to incorporate them (if the core developers choose to do so) and maintain them. Developers who don’t contribute enhancements back might have to re-apply the fixes and enhancements every time they upgrade to a new version of the original code.

Because of this dependency, and for other reasons, company staff should consider becoming active members of the community that develops the code. Developers can learn a lot about the code by participating, and can help set the direction for future development. Of course, the organization should pay for the time developers spend in community activities. For a company that is serious about using free software, it is not cost-free.

Considerations of Open Source Software from a Customer’s Perspective

Opening code is very beneficial to customers for several reasons, but implies a different relationship between the company and its customers. The customers should understand the benefits as well as the implications of the relationship.

The main benefit to the customer, mentioned earlier in this lesson, is that they can have more trust in the software. They know that it won’t go away. Many proprietary companies shut down or take off suddenly in a new direction, abandoning customers with perhaps only a short time to migrate to a product the customers might not like. Open source software, in contrast, doesn’t depend on any single organization. If the project matters, other people in the community will continue the project when the original developers move away.

Customers can also check the quality and security of open source software. They can determine how easily they can add features of their own or port it to a new environment.

Because the source code in an open source project is open for all to examine, a wide range of developers can familiarize themselves with it. On a project that has an active community, many people provide support on the forum. Often, it’s easy to hire people to support the software or to administer and use it within the company, because the new employees already know the code.

It’s very reassuring to customers, who depend on code for day-to-day functioning, to know that they can fix a bug themselves or hire someone to do so. An obscure bug that affects only a few customers might take years to be fixed by the core development team, a constant frustration to users of proprietary software. In addition, when the source code is available, a bug and its suggested fix might be identified more quickly.

What about the relationship between the company and the customer? The company can choose to set up a relationship exactly like one offered by a typical proprietary software company, providing the customer with regular updates and a support contract. The customer never needs to look at the source code or join the community forums.

But most customers would benefit from the unique opportunities offered by open source projects. They can allow their own developers to contribute both information and code. Such participation deepens their staff’s understanding, helps them recruit new staff, and gives them a voice in future development.

Cost Structures and Investments

Programmers are in high demand, so any software effort is going to be expensive. People who know major open source projects are even more in demand, because those code bases are used so widely.

An open source project offers potential cost savings because different organizations and individuals can combine their efforts in a common code base. But running such a project introduces new costs.

If a company opens a project that was developed in-house, the company needs to invest in making it serve a larger (possibly worldwide) community. Some functions that met the company's narrow business needs might need to be rethought so they serve other businesses as well.

If the code is poor or awkward, a company probably shouldn't open it. Potential contributors as well as potential customers will be repelled by the quality. It behooves developers to fix these problems anyway, because poorly coded software is brittle: It is hard to update with new features and tends to develop complex bugs that are hard to fix. These problems are called *technical debt*, and the sooner they are fixed, the better it is for all users.

Finally, it's surprising how often a company has embedded trade secrets, passwords, personal references to individuals, or other sensitive information in code. Developers have to invest time in stripping out such vulnerabilities. Passwords, API keys, certificates, and cloud access credentials should never be in code anyway; they should be managed externally through a secure service.

Let's say that a company has opened its code, and is hoping to benefit from outside contributions. Some customers will pay developers for maintenance and new features. Others will put their own developers on the project, but the core development team has to allocate time to support them. The core development team needs to educate outsiders about the code and the associated coding standards. This team should also guide outside contributors about what to add and where to send back comments on their code.

Keep an eye out for outsiders who show talent. These individuals could be valuable recruits to the core team. They might like to join the team, and they come with considerable knowledge.

When one team of developers is paid, while other people are volunteering their time, the volunteers might feel exploited or ask why they should help. Each contributor, whether an individual volunteer or an organization, has to undergo the types of thinking described earlier in this lesson to decide why they are contributing.

Free code is not cost-free, even if entails no licensing costs. It is simply a different model for development.

Guided Exercises

1. How does open source code improve customer trust in the software?

2. Why are companies tempted to try an open core business model, and why does the model fail to deliver the benefits of open source software?

3. What is self-hosted distribution?

4. What are typical types of help that developers give to outside contributors to their open source projects?

Explorational Exercises

1. You are considering basing a proprietary product on an open source project. What factors would you consider in order to make the decision?

2. You have built several products, for which you charge a subscription, on top of an open source project. What will you do if the open source license requires you to contribute all your code back to the project? Also, you added support for some new communication protocols to the open source project to support your proprietary needs. If you have a choice, will you ask the open source project to integrate this protocol support into their core code?

Summary

In this lesson, you learned the value of opening source code. You reviewed the common business models in use today, and the importance of understanding the impact of the code’s license. You read about issues to consider when incorporating open source code into your enterprise, and how open source benefits your customers. You also learned how costs of open source development differ from those of proprietary software.

Answers to Guided Exercises

1. How does open source code improve customer trust in the software?

   Customers can check the quality and security of the code. They can also trust that they can continue using the code if the original developers stop supporting it.

2. Why are companies tempted to try an open core business model, and why does the model fail to deliver the benefits of open source software?

   At first glance, it seems like open core should offer all the benefits of open source software, while allowing a company to still use a proprietary business model that sells a license to use the software. In practice, an open core business model fails to deliver the advantages of open source, while still having all the increased costs of open development.

3. What is self-hosted distribution?

   Self-hosted distribution means a version of software that can be run on a customer's own equipment.

4. What are typical types of help that developers give to outside contributors to their open source projects?

   Developers on a project typically educate and mentor outside contributors, and check that their contributions meet the team's quality and coding standards.

Answers to Explorational Exercises

1. You are considering basing a proprietary product on an open source project. What factors would you consider in order to make the decision?

   First, decide whether the open source software meets your needs. Check its quality, security flaws that have been publicly reported, and the health of the community around it. Check the license carefully to see whether it requires you to share your modifications to the code. Determine how much you want to participate in the open source project's community and how you'll define the roles that your developers will play in that community.

2. You have built several products, for which you charge a subscription, on top of an open source project. What will you do if the open source license requires you to contribute all your code back to the project? Also, you added support for some new communication protocols to the open source project to support your proprietary needs. If you have a choice, will you ask the open source project to integrate this protocol support into their core code?

   If the project does not require you to share code changes, you probably won't do so because you can charge a subscription more easily for a proprietary product. If you do have to share your code, offer a subscription to a self-hosted distribution. Even if you don't need to share your code, you probably want to ask the project to integrate your support for communication protocols so that you don't have to re-implement that support each time you install a new release of the open source code.

054.2 Service Provider Business Models

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 054.2

Weight

2

Key knowledge areas

- Understanding common business models and revenue streams for organizations providing services related to open source software and open content
- Understanding the impact licenses have on a service provider business models
- Understanding service level objectives and service level agreements
- Understanding the need for security and privacy protection
- Awareness of cost structures and investments needed for open source software service business models

Partial list of the used files, terms and utilities

- Hosted services
- Clouds
- Consulting
- Training
- Hardware sales
- User support
- Terms of Service (ToS)
- Service Level Objectives (SLO)

- Service Level Agreements (SLA)
- Data processing agreements

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	054 Open Source Business Models
Objective:	054.2 Service Provider Business Models
Lesson:	1 of 1

Introduction

Service providers are the backbone of any company or organization that operates in any way via a network. From the Internet Service Provider (ISP), which provides access to the global internet, to specialized applications such as e-mail or customer management, it is software-based services that make actual business operations possible.

Open source software is a component of many different service provider business models. For example, an open source operating system could form the basis of a hosting service business. An open source orchestration tool could be the backbone of an Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS) business, such as a public cloud provider or container platform. An open source application could be delivered online as Software-as-a-Service (SaaS). Some service providers also offer supplementary services related to open source software, such as consulting, training, or customer support.

Revenue Streams

When the service is based entirely or in large part on open source software, we can call it an *open source business model*. Because open source software can be downloaded and used without paying a fee, open source service business models have developed specific products and services depending, among other things, on the proportion of free software in the actual product.

If the entire software is under a free license, a business model can consist of *hosting*, i.e., providing the software on a reliable and secure platform. This eliminates the effort and risk of customers operating their own servers. In return, customers pay the service provider fees for use. The fees may be related to the number of user accounts, the volume of data that is stored or transferred, or certain time intervals. In some cases, multiple different providers offer paid services to the same open source software.

This popular business model often involves *subscriptions*, where customers pay a monthly or annual fee for access to the service. Access is often combined with additional services or functions. Providers may offer multiple tiers of subscriptions, charging a higher rate for more features, more users, more comprehensive services, or a stronger guarantee of reliability. Some providers may also offer a basic level of the service for free, a variation on the “freemium” model.

Another popular revenue stream for the open source service provider business model has always been to offer *support*. Support includes advice, documentation, and training for users who have problems using the software. If the customer prefers to host the free software on their own servers, installation in particular can often be difficult because the user may need to install other supporting tools and libraries, may not know where to put them to make all the pieces work together, or may run into problems on their particular hardware and operating system. Therefore, support and training from a service provider is valuable.

Extra services, such as additional functional or security-related features, can also be developed and provided on a customer-specific basis. Such features or adaptations of the software to a customer’s very specific requirements are a sophisticated and lucrative addition to the business model. It is then up to the service provider and the customer to decide whether the additions flow back into the basic project as free software or become proprietary.

A few open source projects offer a *dual-license model*. The software is available under a free software license, which serves many users' needs. But some users want to incorporate the software into a proprietary product, which cannot be done under a reciprocal license. Therefore, a standard copyright-based license is provided for these users. The dual-license model is most effective for databases, because many companies would like to offer a feature-rich and efficient database as part of a proprietary software product.

In an *advertising revenue model*, customers do not pay for access to the service. Instead, the service provider displays ads to the customers using the service and charges companies to place their ads. Open source service providers often avoid using proprietary ad services due to concerns about user privacy. However, alternative open source ad services do not engage in invasive ad tracking.

In a *commission-based model*, the service provider receives a percentage or fee for managing transactions through their online service. This model is the basis for most e-commerce sites, travel booking sites, and payment processors. Many open source projects in this space are general-purpose platforms intended to be either self-hosted by the service provider or offered as a secondary commission-based service, where the customer-facing website pays a percentage or fee for a hosted version of the open source platform.

An *open content business model* involves creating and distributing content under licenses such as Creative Commons so others can freely use, modify, and share it. Users are encouraged to contribute to the content, improving it over time. While the content itself is free, revenue can be generated through complementary products or services such as donations, displaying ads, offering freemium features, selling related merchandise, or providing consulting, training, or support services.

Choosing Between Hosted and Self-Hosted Services

Hosted services are ideal for organizations looking for ease of use, lower initial costs, scalability, and professional security management without the need for deep technical expertise. However, these services come with potential downsides in terms of control, customization, data privacy, and dependency on the provider.

Hosted services usually require minimal setup, and the service provider handles maintenance, updates, and system security. They often make global data centers available, providing better performance and reliability for international users. Service providers can scale resources up or down based on demand with minimal effort from the customer. Moreover, hosted services often have dedicated security teams to manage and respond to threats, and service providers may offer compliance with various regulatory standards.

On the other hand, hosted services store customers' sensitive data off-premises, which may raise concerns about data privacy and trust. A shared hosting environment can pose additional security risks. The service provider’s outages or downtime directly affect the customer’s service availability. Depending on a single provider can be problematic if the provider changes terms or discontinues the service.

Self-hosted services are thus suitable for organizations with the necessary technical expertise and resources to manage their infrastructure. However, self-hosting involves higher initial costs, ongoing maintenance, and scalability challenges. The choice between hosted and self-hosted services ultimately depends on each customer’s needs, technical capabilities, budget, and priorities for control, customization, and data privacy.

Impact of Licenses

Basically, open source software licenses work well with service provider business models because the customer doesn’t expect to own the software that provides the service. Instead, they’re paying for access to the service.

On the other hand, the use of open source software in services such as IaaS, SaaS, or PaaS raises legal issues that are often not clarified. Only a few licenses, notably the *GNU Affero General Public License*, explicitly regulate the use of free software “in the case of network server software.”

With other popular licenses — both the more restrictive copyleft licenses such as the GNU General Public License, and permissive licenses such as the BSD licenses — there is room for discretion as to what can be considered "'copying,`" “conveying,” or “distribution” when the software is used over the network. Decisions about whether and in what form the source code is to be made available to the application or attribution requirements depend on such definitions. The connection with proprietary software components or the licensing of self-developed supplementary components must also be legally clarified. For these reasons, it is crucial in the context of service provider business models to clarify licensing issues relating to the use of free software.

It is good practice for service providers to publicly list all the open source software they use together with their respective license, even when they aren’t legally required to do so.

Considerations of Security and Privacy Protection

The success of a service provider business model relies heavily on a satisfying customer experience, especially with open source software, where customers have the freedom to self-host the software or choose a competing service provider for the same software. Thus, the main competitive advantage of an open source service provider is the customer experience they offer, which may be more convenient, cost-effective, reliable, secure, or performant than self-hosting. However, the trade-off is that customers must share data with the service provider, which may introduce privacy concerns for personal or sensitive data.

For security, customers should ensure that the service provider has a robust process for promptly applying security patches. It’s worth evaluating how the service manages dependencies on other open source projects to ensure all the software is up-to-date and secure. The customer should also assess the provider’s security policies, including how they handle data encryption, access controls, and incident response.

The transparency of open source software allows for thorough code review by the community, which can enhance security by identifying and fixing vulnerabilities. So customers should look for reviews or audits of the software by reputable third parties or security experts within the community. These audits should confirm that the provider follows secure coding practices and standards during the development process, and should consider whether the provider uses CI/CD pipelines with integrated security checks to catch vulnerabilities early.

For privacy, customers should verify that the service collects and stores only the data necessary for its operation, minimizing the risk of data exposure. The service should encrypt data both in transit and at rest, using strong encryption. Access control mechanisms should ensure that only authorized personnel can gain access to sensitive data.

The customer should check that the service gives users control over their data, such as the ability to delete or export it. The service should comply with relevant privacy regulations, such as the European Union’s General Data Protection Regulation (GDPR) and California’s Consumer Privacy Act (CCPA), and provide transparency about its data handling practices.

It’s important to check that the provider has clear policies and procedures for responding to data breaches, including notification requirements. The customer should also review the provider’s privacy policy to understand how data is collected, used, shared, and protected, and consider any third parties the provider may share data with and ensure they also adhere to strict privacy standards.

In general, it’s valuable to look at community feedback and reviews to gauge the reputation and trustworthiness of the software and the service provider. A robust community or organization should be actively maintaining and supporting the open source project. The customer should verify that the provider has regular data backup procedures and robust disaster recovery plans, and check whether the service uses data redundancy to ensure availability and reliability.

Service providers should be aware that customers will be evaluating their security and privacy practices, community reputation, compliance with regulations, and transparency in handling data, and consequently should take steps to follow best practices.

Common Agreements Between Service Provider and Customer

Service providers commonly adopt legal terms and agreements that form the legal and operational backbone of the relationship between the company and their customers. These agreements help ensure clear communication, set expectations, define responsibilities, and provide legal protections for both parties.

The *Terms of Service* (ToS) for a service, also known as *Terms and Conditions* (T&C) or *Terms of Use*, is a legal agreement between a service provider and the customers or users of that service. The ToS outlines the rules, responsibilities, and limitations that govern the use of the service. It protects both the service provider and the customer by clearly defining what each party can and cannot do while providing or using the service. Some common elements of a ToS agreement are confirmation that the user agrees to comply with its terms, guidelines on acceptable behavior and prohibited activities, details on who owns the content created or uploaded by users, information on account creation, security, and termination, arbitration or mediation processes for resolving conflicts, and limits on the service provider’s liability for damages.

A *Privacy Policy* is a statement that outlines how the service provider collects, uses, stores, and protects user data. Some common elements of a privacy policy are the types of data the service provider collects and their collection methods, how they use the data, conditions under which the service provider may share data with third parties, and information on user rights regarding their data, such as access, correction, and deletion.

A *Service Level Agreement* (SLA) is a formal, documented agreement between a service provider and a customer that outlines the expected level of service, including the specific performance metrics, responsibilities, and expectations. The SLA defines the standards for service delivery, providing a clear framework for measuring and managing service performance. Some common elements of an SLA include a detailed description of the services provided, specific targets for service performance such as uptime and response times, methods for measuring and reporting performance, consequences for failing to meet performance targets, and procedures for reviewing and updating the SLA.

A *Service Level Objective* (SLO) is a key component of a Service Level Agreement that specifies measurable targets or goals for service performance. These objectives quantify the expected level of service between a service provider and a customer and are used to ensure that the service consistently meets the agreed-upon standards. Some common elements of SLOs include specific metrics the service provider will measure to evaluate the service performance (for example, uptime, response time, resolution time, and throughput) the desired or expected values for each performance metric, the techniques and tools used to measure and monitor the performance metrics, the specific components or aspects of the service that the SLO covers (for example, hardware, software, network components, or specific processes), and the consequences or rewards based on whether the service meets or fails to meet the SLOs (for example, financial penalties, service credits, or performance bonuses).

A *Data Processing Agreement* (DPA) is a contract between a customer (the data controller) and a service provider (the data processor) involved in processing personal data. The DPA details the responsibilities and obligations of both parties to ensure compliance with data protection laws, such as the GDPR. In some jurisdictions, a DPA between a service provider and a customer is mandatory.

Cost Structures and Investments

The most significant expenses in a service provider business model are the fixed costs related to hosting the services, such as purchasing server hardware, paying for data center space, electricity to power and cool the servers, network bandwidth, and salaries for deployment and maintenance staff. Some service providers may also have variable costs for software licenses or third-party services. Customers who choose to self-host their services bear these costs directly.

By choosing open source software, service providers and customers can reduce or eliminate licensing costs compared to proprietary software, because by definition open source software charges no fee for the software license. Employing open source software to build services can also save on development time and resources, and community-driven maintenance and updates can lower long-term costs. However, it’s important to consider the total cost of ownership (TCO) of the open source software, including potential support, maintenance, and integration costs.

Guided Exercises

1. Name some ways that service providers can earn revenue without charging the users of the service.

2. What are some reasons that a service provider might offer its software for self-hosting by customers?

3. Why would a service provider release its software under the GNU Affero GPL license?

Explorational Exercises

1. What is an Acceptable Use Policy (AUP)?

2. What is an End User License Agreement (EULA)?

Summary

This lesson discusses service provider business models that rely on open source software, and various revenue streams, including subscription, ad revenue, hourly rate, and commission-based models. It highlights the impact of open source software licenses on service providers, emphasizing their benefits and obligations. The considerations of security, privacy, and reliability of services, along with the importance of Terms of Service (ToS), the Service Level Agreements (SLA), and the Data Processing Agreement (DPA) are also addressed.

Answers to Guided Exercises

1. Name some ways that service providers can earn revenue without charging the users of the service.

   The service can offer ads, for which the advertisers pay. The service can charge a commission to vendors who offer products or services on the site, such as retail or travel. The service provider can sell customer data, a practice that many customers would object to and that should be explained in the Terms of Service.

2. What are some reasons that a service provider might offer its software for self-hosting by customers?

   Potential customers can try out the software to check its features and reliability before signing up with the service provider. Customers might also find bugs and even propose fixes.

3. Why would a service provider release its software under the GNU Affero GPL license?

   The Affero GPL requires other service providers to release any changes they make to the software under the same license. This provision prevents competitors from taking advantage of the service provider by adding a few enhancements that they keep proprietary and then promoting their version of the service as superior to the original.

Answers to Explorational Exercises

1. What is an Acceptable Use Policy (AUP)?

   An Acceptable Use Policy (AUP) is a policy that defines acceptable and unacceptable uses of the service, to prevent abuse and ensure a secure and reliable environment. Some common elements of an AUP include a description of specific actions that are not allowed, such as spamming or hacking, obligations of users to use the service responsibly, actions the service provider may take in case of violations, such as suspending a user’s account, and procedures for reporting and addressing violations.

2. What is an End User License Agreement (EULA)?

   An End User License Agreement (EULA) is a legal contract between a proprietary software provider and an end-user (not a company or organization), granting the user the right to use the software under certain conditions. Some common elements of EULAs include the scope and limitations of the license (for example, personal use or non-transferable license), prohibited actions (for example, reverse engineering or redistribution), conditions under which the license can be terminated, and ownership and protection of intellectual property rights. Fully open source services substitute an open source software license in place of a proprietary EULA.

054.3 Compliance and Risk Mitigation

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 054.3

Weight

3

Key knowledge areas

- Understanding how to ensure license compliance
- Understanding how to maintain information about licenses
- Understanding the concept of Open Source Program Offices
- Understanding the implications of copyright, patents and trademarks on open source business models
- Awareness of legal risks related to open source business models
- Awareness of financial risks related to open source business models

Partial list of the used files, terms and utilities

- Software Composition Analysis (SCA)
- Software Bill Of Materials (SBOM)
- Software Package Data Exchange (SPDX)
- OWASP CycloneDX
- Open Source Program Offices (OSPO)
- Product warranty
- Product liability
- Export regulations

- Impact of mergers and acquisitions

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	054 Open Source Business Models
Objective:	054.3 Compliance and Risk Mitigation
Lesson:	1 of 1

Introduction

An old quip in open source says that “Free software doesn’t come for free.” While free and open source software permit great innovation and creativity, users and developers have to comply with a number of rules. This lesson covers compliance and risk in open source software.

The lesson lists basic steps that developers need to comply with licenses, the risks of using open source software, and ways of tracking and cataloguing the software the organization is using. We’ll look at how these tasks can be turned into policies and promoted by an Open Source Program Office (OSPO).

Requirements for Releasing Software Based on Open Source Components

If the organization uses software internally, free and open source software licenses impose no requirements. The organization might define its own rules to prevent vulnerabilities, make sure everyone is using the same components, or for other reasons. But that’s beyond the scope of this lesson. Any requirements that the open source licenses or the organization itself impose on the use of software should be documented, and the organization should provide training in its policies.

Even if the organization runs software on its web server and offers services with which people outside the organization interact, most free and open source software licenses impose no requirements. However, the GNU Affero Public License requires adherence to copyleft requirements for software running as a service too.

The legal requirements of each major free and open source software license have been fully explored in other lessons, so this section just offers a list of actions that developers and managers should take to comply:

Vetting open source components

The organization needs clear policies and education around what open source software to use and where to include it in products. Such policies cover compliance, as well as other issues such as ensuring security and participating in the community that develops the software.

Keep the original license in the code

Developers must not strip the license from the component they use. Including the license in the source code is usually a requirement in the license. In fact, removing the license could be considered plagiarism, because the developer makes it look as if they wrote the code. Removing the license could also lead to serious trouble later, because the organization could violate the license when the code is included in its products.

Keep track of licenses

The organization must maintain a catalog showing the license of each file or package used by the organization, in order to make sure it is complying with the licenses.

Acknowledge the author

Nearly all licenses require that credit be given to the copyright owner (often called the “author”) when discussing and promoting the software. Each license explains what to include in this notice: Typically, they require a standard copyright notice with the year and the name of the copyright owner. This notice should appear in any documentation related to the product using the component, including the advertising and web site for the device or program.

Don’t imply endorsement

Some BSD licenses require the person distributing a product with the components to make sure not to imply that the copyright owner endorses the product. Even if the requirement is not stated, complying with it is the ethical thing to do.

Offer the source code for copyleft licenses

If an organization distributes a binary file containing copylefted components, whether as a software distribution or on a device, the organization has to offer the public the source code to those components. If the organization changes the code and redistributes the derivative work in binary form, the source code for the changed (*derived*) version must be offered to the public as well. Distribution can be done through any means that make access convenient to the public, such as posting the code on a web site or offering it on a disk or USB stick.

Don't include copylefted components in proprietary products

If the organization incorporates copylefted components into a product, under some circumstances the organization has to release the whole product under the same copyleft license. The conditions that trigger this requirement vary from license to license. It's sometimes unclear, though, what kind of use triggers the copyleft requirement. So, except for clear situations such as using an open source library (which should not trigger the copyleft's reciprocal requirement), developers should be very careful about their use of copylefted components unless the organization is prepared to release its product under the same license.

Document code changes

When distributing modified versions of code, clearly indicate the changes that the organization made.

Grant patent rights

Some free or open source software licenses require a grant for patent use on the software. Thus, if an organization releases code under such a license, and holds a patent on some process in the code, the organization can't charge a fee or exert other controls based on its patent to anyone using or adapting the code.

Respect trademarks

Some open source software is covered by trademarks. Trademarks can cover words and phrases, logos and other images, and many other things. On the one hand, an organization must handle the trademark properly for any software it uses: A common violation is to parody, alter, or simply display a trademark without permission. On the other hand, if the organization wants to use the trademark, it must comply with the trademark owner's requirements, which might rule out modifications to the software.

Risks of Open Source Software

This lesson is about compliance, so we'll focus on risks in that area, but cover a few other topics too.

License Violation

Free and open source software licenses have to be treated as seriously as other licenses and contracts. Organizations do enforce them, such as the Software Freedom Conservancy, which acts on behalf of small copylefted projects that don’t have their own enforcement mechanisms.

These organizations usually approach lawsuits as a last resort. Most violations are unintended, and are easy to resolve with education. Still, it hurts an organization’s reputation to be seen as ignorant and insensitive toward the communities on whose software it depends.

And lawsuits have actually been launched when a software user is flagrantly refusing to cooperate, and when the software and the defendant are important enough.

Even if an organization is not punished with a lawsuit, the damage that a license violation does to its relationship with the community, as well as to its reputation, can be substantial. A project can be derailed by something as simple as having developers' questions go unanswered on the forums devoted to the software they’re trying to use.

It can be a disaster for someone to find copylefted software in a proprietary product. To comply with the license, the developers must either strip out all the copylefted code or free their product under the copyleft license. Making their software free, with source code available, could have dire effects on business models based on license fees or other ways of monopolizing the value of the product.

Trust and Reputation

We’ve seen that license violations can be very damaging. Other types of behavior that disrupt trust and reputation also introduce risks.

Some organizations release software under a free or open source software license, and then announce at some point that future versions will be under a proprietary license. This switch can be tempting, because many open source projects fail to receive financial support from enough clients.

But such license changes induce anger among both clients and the external developers contributing to the project. Sometimes, external developers take the free version and continue to develop it independently, a step known as *forking* the project. The free version might become competitive with the company’s proprietary version and draw customers away from the company.

There are also risks when participating in project forums. A company must learn how to be a good citizen. Common problems include:

- Trying to use a company’s financial or code contributions as leverage to force the project in a direction that other developers don’t want
- Making too many demands on the community, such as by pressuring it with a lot of feature requests or even too many questions
- Being rude in other ways and violating the project’s code of conduct
- Trying to dominate publicity or capitalize inappropriately in other ways on the project’s success

Unrewarded Investments

Business models are discussed in another lesson; this section just notes the financial risk of participating in open source projects.

Companies might start or join open source projects with the expectation of deriving revenue through some mechanism such as support contracts, SaaS contracts, data collection, or even donations and grants. Unfortunately, those who conduct open source projects often find them less lucrative than hoped for. Of course, any business takes a risk when starting a new project, but it’s particularly hard to find a reliable source of income for open source.

Open source seems most sustainable when it supports some other form of income, such as the sale of hardware devices, cars, printers, etc.

Forks

As we’ve seen, members of a project sometimes disagree on the project’s roadmap, leadership, or other aspects and create an alternative project on the same code base. Forking can also be done by an organization to create a specialized version of the software. For instance, Android uses a specialized version of Linux that Google maintains. (Google also makes a lot of contributions to the core version of Linux, and they are not always accepted.)

Organizations may create a fork for various reasons. They might submit their changes to the core project and have them rejected, because other developers don’t want them. In a project with a permissive license, or a project used only within the organization, it might want to keep its changes secret.

The risk of a fork is that the developers of the forked project are responsible for the maintenance of the entire product. If important bug fixes or new features are added to the core project, the developers of the forked project must duplicate the changes or forego their benefits. As time passes and the projects get further apart, keeping apace with changes in the core project becomes harder and harder.

Incompatible Licenses

As explained in other lessons, some free and open source software licenses have incompatible clauses. Such components can’t be used together in a product.

This problem is likely to arise during a merger or acquisition. Each company might have been using software with a particular license, and if they want to combine the code in their projects, they must make sure the licenses are compatible.

Product Liability

Many open source software licenses (and the terms of service for many proprietary software and services, for that matter) explicitly disavow any liability for the software’s use. Another way to say this is that the license or terms of service offer no warranty or guarantee.

Software vendors are rarely held responsible for problems with their software, but clients occasionally sue them or try other forms of punishment. Courts don’t have to recognize the “no warranty” clauses.

Increasingly, laws and regulations are imposing responsibilities on the creators of software. These legal restrictions — or at least, proposals for restrictions — are seen particularly now for artificial intelligence, but sometimes are broader.

Export Regulations

Many countries restrict the export of goods and software. In software, such regulations apply most often to security products, and specifically to the use of cryptography. The United States used to have strict controls on any software containing cryptography, but the controls have been relaxed on open source projects.

Companies should be aware of the export regulations where they are creating software, in case these regulations affect the sale and distribution of their products.

Software Bill of Materials: Know What You’re Using

Many products come with a bill of materials, which is just a list of all their components. For instance, when you buy a consumer product, it might include a sheet of paper listing all the parts, down to the nuts and bolts. The list might include useful information such as the dimensions of a part, and a part number so you can order a new part.

Open source projects are now including a *Software Bill of Materials* (SBOM, pronounced “S-bomb”). At the very least, an SBOM lists packages, filenames, licenses, authors or owners, and version numbers. Many SBOMs go further and include information about security vulnerabilities.

Projects generate an SBOM for each release using automated tools. Users can scan the SBOM to find information they need. For instance, extracting licenses helps the organization decide quickly whether the components are compatible with their own code and whether it’s legally permitted to use the components with their code.

Similarly, extracting version information on each package helps the organization discover whether different parts of its product depend on different versions of a particular library. Using two versions of the library causes bloat at the very least. It also could create confusion and errors, if a component is built with the wrong version.

Standards for SBOMs

Because computing environments use enormous quantities (sometimes tens of thousands) of components and make frequent changes, SBOMs must be highly structured so that information can be extracted automatically and quickly. This section describes two of the most popular formats in the open source world:

Software Package Data Exchange (SPDX)

The format represents each package and all the contents of that package in a tree structure. The format documents dependencies between files, and other relationships. Pointers to information, known as “snippets,” can be created and then used throughout the document so that information needs to be defined in only one place. This format was developed by the Linux Foundation.

CycloneDX

This is a larger format with more detailed fields, particularly for vulnerability information. The format was created by the *Open Worldwide Application Security Project* (OWASP) and is popular in the military and other organizations with a focus on security. For instance, an entry for a file could include a name for a security vulnerability, the source of information about the vulnerability, affected targets, and more. This format is also designed for cloud deployments that can include thousands of different systems.

Both SPDX and CycloneDX create hierarchical structures in various formats, including JSON and XML. Many tools exist for each standard, both to create the formats and to scan created files for information. Popular version control repository sites let developers create an SBOM in one of these formats with the click of a button.

Software Composition Analysis

Knowing what an organization is using requires an assessment of its software that includes an understanding of where it comes from, what vulnerabilities it contains, what licenses it uses, and other factors that help an individual or organization decide whether to use the software. This task is called *Software Composition Analysis* (SCA), and many tools exist to perform sophisticated scans of SBOMs and of software itself.

Some tools extract license information, which helps an organization decide whether the software is safe to include in a product and whether different components have compatible licenses. Some tools even compare code snippets to libraries of common open source projects to find out whether code was taken from the open source projects without proper attribution.

Running these tools is especially crucial during a merger or acquisition. The acquiring company might find that the software it wants to acquire is less valuable than they had expected, because it contains open source components that impose requirements interfering with its intended use in the new organization.

Formal Policies and Compliance

The processes and techniques described in this lesson should be fashioned by managers with input from developers, attorneys, and other knowledgeable people. This section describes some of the policy decisions that organizations need to make concerning open source software. We'll end with a description of an *Open Source Program Office* (OSPO), which can be an advocate and information source for the policies.

Principles Governing the Use and Contribution of Open Source Software

Organizations can benefit greatly from using and contributing to open source software, but they need to do so in ways that protect their own interests as well as benefit the open source projects. Policies should be defined for:

- Where to use open source software (operations, internal projects, client-facing products, etc.)
- What makes an open source project appropriate for the organization: features, performance, security, roadmap for future extensions, and community viability
- Scans for software composition analysis, and where resulting documentation should be stored
- Rewards for developers contributing to open source software, including their participation in public forums
- Guidelines for contributing to projects, including how to be a public representative of the

company

- Documentation requirements, so that developers outside the core team can understand how to contribute and interact

The OSPO can coordinate the creation of these policies and education to ensure compliance.

Contributor Agreements

Open source projects need to ensure that contributions are legitimate: for instance, that the code has not been taken from some proprietary product or an open source project with an incompatible license. Many open source projects require developers to provide a document called a *Contributor License Agreement* (CLA) to ensure the legitimacy of their contribution.

Developers at an organization who contribute to these projects might ask the organization's lawyers to review and approve the CLA. Thus, the lawyers should understand the relevance of the clauses of CLAs and be prepared to vet them.

Numerous criticisms have been aired about CLAs, including their complexity and the loopholes they leave for projects to use contributed code in ways not desired by the contributors.

Many projects, instead of a CLA, ask a developer to sign a short document called a *Developer Certificate of Origin* (DCO), attesting that the developer has a right to contribute the code. The DCO, which is discussed in another lesson, generally is not submitted to a lawyer for review.

Some projects simply ask contributors to sign the copyright to their code over to the project in a *Copyright Assignment Agreement* (CAA). However, many contributors dislike this practice, because they can't use the code in some proprietary project of their own and because it grants a lot of power to the project.

The Open Source Program Office (OSPO)

Open source projects combine unusual social, technical, legal, and organizational factors. At this time, knowledge of these factors has not swept through organizations to the point where all managers, developers, attorneys, and others understand them deeply. So organizations can benefit greatly from creating an *Open Source Program Office* (OSPO) as a central point for advocacy, policy-making, policy enforcement, and education.

As a kind of all-purpose department, OSPOs can perform many roles, such as:

Promoting open source

OSPOs can disseminate both the concepts behind the open source movement and suggestions for using open source throughout their organization. They can remind developers to look for open source solutions to the problems they're solving and encourage them to adopt appropriate software. They can urge managers to grant time to developers to participate in open source projects, and to recognize that participation when evaluating developers for salary raises and job promotions.

Policy definition

OSPOs can mobilize managers to create policies and set up repositories for them.

Policy enforcement

OSPOs can remind developers to scan software and SBOMs and to follow corporate rules about the use of open source. The OSPOs can preserve documentation about the software adopted, and other aspects of corporate use.

Education

OSPOs can explain to developers how to evaluate and participate in open source projects, explain to lawyers the details of licenses and other legal considerations, and help the company understand the organizational and cultural changes that facilitate the use of open source software.

There are numerous documents and online resources for creating an OSPO. A small organization can farm out the task to a third party who is expert in that area.

Guided Exercises

1. Why shouldn’t a developer just take code they like from an open source project and put it into their own code without preserving the license?

2. What sorts of documents would a developer sign when making a contribution to an open source project?

3. You find some open source software that would make an excellent basis for your retail site. Can you overlay your logo over its popular logo to make an eye-catching image for your web site?

Explorational Exercises

1. What considerations might lead you to make a fork of an open source project for your own use, despite the drawbacks of forks?

2. When you find some open source software that meets your need, what are some reasons you might reject it?

3. You'd like to use a copylefted logging tool in your proprietary product. Is there a way to combine them that doesn't require you to offer your product under the copyleft license?

Summary

This lesson has covered many considerations to take into account before using free and open source software. It explained how to adhere to licenses, various legal, reputational, and financial risks, and how to define important policies and enforce them within your organization.

Answers to Guided Exercises

1. Why shouldn’t a developer just take code they like from an open source project and put it into their own code without preserving the license?

   This is usually a violation of the open source license, and also represents plagiarism and copyright infringement. Practically speaking, the organization can be forced to comply with the license, with consequences ranging from embarrassment and reputational damage to the destruction of their business model if copylefted code is mixed in with proprietary code.

2. What sorts of documents would a developer sign when making a contribution to an open source project?

   A Contributor License Agreement is a legal document granting rights to the open source organization to use the code. A Developer Certificate of Origin is a shorter and simpler document promising that the developer has a right to contribute the code. A Copyright Assignment Agreement grants all rights to the receiving organization.

3. You find some software that would make an excellent basis for your retail site. Can you overlay your logo over its popular logo to make an eye-catching image for your web site?

   If the project has trademarked its logo, your change would probably violate the trademark. Even if the logo is not trademarked, your change could be seen as disrespectful as well as confusing.

Answers to Explorational Exercises

1. What considerations might lead you to make a fork of an open source project for your own use, despite the drawbacks of forks?

   If you are satisfied with the current state of the code, you might not need to keep up with changes to the core project. You might want to create a product substantially different from the uses to which the core project is put, and therefore be willing to part ways with the core project. The code might be valuable enough in your business plan that your team is willing to take on complete responsibility for its development and maintenance.

2. When you find some open source software that meets your need, what are some reasons you might reject it?

   The code might contain too many bugs and security vulnerabilities, the project's developer community might not function well, you might not like the direction in which the code is evolving, or the license might not be compatible with other code in your product.

3. You'd like to use a copylefted logging tool in your proprietary product. Is there a way to combine them that doesn't require you to offer your product under the copyleft license?

   Integrating the code for the tool into your code might trigger the copyleft's reciprocal requirement, depending on which license is in effect. The logging tool must be separated from your product so that your product is not seen as derivative. For instance, it might be safe to run the copylefted tool as a separate process and communicate with it through message passing.

Topic 055: Project Management

055.1 Software Development Models

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 055.1

Weight

3

Key knowledge areas

- Understanding the relevance and goals of project management in software development
- Basic understanding of waterfall software development
- Basic understanding of agile software development, including Scrum and Kanban
- Understanding the concept of DevOps

Partial list of the used files, terms and utilities

- Phases in waterfall projects (requirement engineering, business analysis, software design, development, testing, operations)
- Roles in waterfall projects (project managers, business analysts, software architects, developers, testers)
- Organization of Scrum projects (sprints and sprint planning, product and sprint backlog, daily scrums, sprint review and sprint retrospective)
- Roles in Scrum projects (product owners, developers, scrum masters)
- Organization of Kanban projects (Kanban boards)

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	055 Project Management
Objective:	055.1 Software Development Models
Lesson:	1 of 1

Introduction

Life is full of projects. A project might be planning a movie night, a trip, or a family event, scheduling the children’s week, or improving your work on a hobby. It does not matter what you want to carry out: You need to make plans and take actions. We usually create several scenarios, with plan B, plan C, etc. This is no different in the world of software development.

Roles in Software Development

A variety of skills are needed for successful project management, so it’s useful to find different people to fill well-defined roles. The following sections lay out some of the roles commonly found on software projects.

Project Manager

Managing a project is a highly complex task. Some situations look easy at first — but it takes care and experience to get them right. The person doing this is the *project manager*.

Project manager responsibilities include ensuring that the project is finished on time, within budget, and with acceptable quality. Even if they don’t write a single line of code, project managers are the ones who are responsible and accountable for the team’s results. They have to align with clients about deadlines. They also talk to the people in other roles to make sure everything is in order.

Business Analyst and Requirement Engineer

There is no place for micromanagement — at least not in a healthy workplace. *Business analysts* (BA) and *requirement engineers* (RE), internal or external to the project team, are the right hands of the project managers. They are responsible for creating rich and clear documentation about the requirements. They are also the bridge between the clients and the developers. The clients communicate through plain language, which the BA/RE ensures will provide clear and unequivocal documentation with which developers can effectively do their work.

Imagine you are being asked to bring a “big cake” to a party. What does “big” mean? Ten slices or twenty? Maybe it’s the cake for a big wedding and it should be one hundred slices. The BA/RE has to specify requirements such as quantities exactly, so that the developers will know, say, that an application is designed for a specific number of users.

We are just at the beginning of defining requirements, but you already can see the importance of clear communication. Communication is essential for any joint work, but is perhaps even more important on an open source project, as people with very different backgrounds might work on each part of the project. Someone may collaborate as a student, someone else as a parent, a person who is retired or between jobs, etc. Even so, progress has to be smooth in this environment — so communication can’t hold up progress.

Developers, Architects, and Testers

To implement the requirements, a software project needs *developers* who are creating the product based on the documentations and design decisions. Their work is judged by the *architect*, who usually has the most extensive technical and domain knowledge of anyone on the project. Every developer, especially the senior ones, is responsible for their own code, but big decisions are made or approved by the architect.

Sometimes people are specifically designated as *testers*, who are responsible for all kinds of testing, documenting the results, and creating issue tickets.

Planning and Scheduling

Now that we have discussed basic roles, let’s talk about the phases of a project: planning, scheduling, implementing, maintaining/testing, and delivering. Some phases differ in various software development models, but planning and scheduling are general topics we'll discuss before diving deeper into different models.

Planning takes place after the BAs/REs provide the list of requirements. Planning consists of one or more meetings about what the team wants to achieve, how they plan to accomplish it, and how long it would take. Usually, some risk analysis takes place at these meetings as well.

Planners then have to *schedule* the work and create *milestones*. At each milestone, the team knows that one big, important part is done. Long-term components or features can take months, so planners have to factor in vacations, holidays, and — especially during cold seasons — some sick time off, to ensure an accurate delivery time and to avoid rush and panic approaching the deadline. Under a good schedule, the developers feel more relaxed and can deliver high-quality solutions in time, without breakdowns.

Common Tools

Two tools that benefit all models, and project management in general, are a *version control system* (VCS) and an *application lifecycle management* (ALM) platform. Version control is discussed in depth in another lesson, so let's say a few words about ALMs.

ALM is a comprehensive framework that manages the lifecycle of a software application from initial development through maintenance and eventual retirement. ALM integrates people, processes, and tools to enhance collaboration and efficiency, ensuring that all stages of the software lifecycle are well-coordinated and aligned with business goals. This approach helps to maintain the quality, performance, and reliability of applications throughout their lifespan.

Now that we have a general overview of roles and lifecycle phases, let's dive into models.

Waterfall Model

This is one of the earliest and most straightforward methodologies in software development. You can imagine it as a staircase, where each step needs to be finished before moving forward. But the model goes in only one direction, making it suitable for projects with clear requirements and minimal or no expected changes.

Although the simplicity of this model can be an advantage, the rigid menthodology can be a drawback when flexibility and adaptibility are necessary. And these days, usually, everything changes while you're working on your project.

As previous sections explained, to start development, a project needs to have the requirements, a finished plan, and a finalized schedule for the work with milestones. In the waterfall model, these are the outcome of requirement engineering and business analysis.

Requirement Engineering

In this initial phase, all the project requirements are gathered and documented. This work involves extensive discussions between the project manager and stakeholders to understand their needs and expectations. The outcome is a comprehensive requirements specification document that outlines what the system should do.

Business Analysis

Business analysts examine the requirements from a business perspective to ensure they align with organizational goals. BAs perform feasibility studies, risk assessments, and cost-benefit analyses to determine whether the project is viable and worthwhile. The insights gained are used to refine the project scope and objectives.

Software Design

Software design is the step where the architect creates a detailed design specification for the developers to follow. Given this specification, the team can start to implement the requirements — in other words, start the developement phase, which comes next.

Development

The development phase is where the magic happens — where the code is written. Following the documentation and the design decisions assiduously, the developers write their parts. After developers review their code with other developers or the architect, this phase can be considered done.

Testing

Development is followed by a phase of testing, managed by the testers. There are different types of testing, described in another lesson, such as unit testing, integration testing, system testing, and acceptance testing.

The goal of these tests is to bring issues to the surface before release, and to allow the developers to fix them in time — not after the project is deployed and published and users can be irritated and frustrated by bugs.

Operations

After the tests are passed and bugs are fixed, the project can be released — that is, deployed to the production environment. This phase can contain installation, configuration, and sometimes even user training. After the project is ready to use, it is in a maintenance phase, where the team can handle issues from the users and deliver updates if necessary.

Evaluating the Waterfall Model

What have you noticed while reading about this model? Do you find it efficient? Is something missing? Let’s see the pros and cons.

The waterfall model benefits from:

Clear structure
: The linear process makes it easy to manage, understand, and follow.

Meticulous documentation
: Detailed and thorough documentation at each phase helps the team understand what’s needed during development and future maintenance.

Predictability
: Clear, defined milestones help to provide a foreseeable timeline and budget.

Simplier project management
: Thanks to its linear and straightforward flow, the model is easy to manage.

The waterfall model suffers from:

Rigidity
: The linear inflexibility of the model makes it difficult to implement any changes after the requirements phase is finished.

Late testing
: During the development phase, testing is haphazard or missing, which can lead to late recognition of important issues.

Assumption of perfect requirements
: The model requires and assumes that every requirement is correct and clear, which can be very unrealistic. The model does not make room for checking during development to make sure the requirements are accurate and are understood by developers.

Lack of feedback

Only in the last phase can a customer say anything about the product. So if something (or a lot of things) do not meet their expectations, the problem turns up only at the very end.

Agile Software Development, Scrum, and Kanban

To explore this set of more modern software development models, let’s start with the word *agile*: What does it mean? Agility expresses the ability to react and move quickly, in both the physical and mental world. The goal is the same in software development.

Agile software development is a methodology that builds on iterative development, flexibility, and collaboration. It focuses on delivering small improvements while always gathering feedback and implementing it during development. This approach allows quick reactions, adjustments, and responses, saving time and ensuring that the project meets users' and clients' expectations.

The agile movement was launched in 2001 by an online *Manifesto for Agile Software Development* that listed its principles as follows:

> We are uncovering better ways of developing software by doing it and helping others do it.
>
> Through this work we have come to value:
>
> - *Individuals and interactions* over processes and tools
> - *Working software* over comprehensive documentation
> - *Customer collaboration* over contract negotiation
> - *Responding to change* over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.
>
> — Manifesto for Agile Software Development

Scrum

Scrum is one of most popular frameworks — maybe the most popular — within agile software development. Scrum is built upon the following building blocks:

> In a nutshell, Scrum requires a Scrum Master to foster an environment where:
>
> 1. A Product Owner orders the work for a complex problem into a Product Backlog.
> 2. The Scrum Team turns a selection of the work into an Increment of value during a Sprint.

> 3. The Scrum Team and its stakeholders inspect the results and adjust for the next Sprint.
> 4. Repeat
>
> — The 2020 Scrum Guide

But who is the scrum master and product owner? What is a product backlog and a sprint? Let's dive into the roles and processes.

Sprints and Sprint Planning

A *sprint* is a development phase typically lasting two to four weeks, during which some previously agreed tasks and/or features are completed. To get agreement on tasks, *sprint planning* comes in. This process is where the team makes decisions about which tasks can be completed during the upcoming sprint.

These choices are based on the priorities set by the *product owner* (PO), who controls the project and often provides the funding for it.

Product Backlog and Sprint Backlog

As just explained, the PO manages the list of priorities, which contains all the desired work on the project. In Scrum, this list is called the *product backlog*. Each *sprint backlog* contains the selected tasks for the current sprint from the *product backlog*. The sprint backlog contains the elements that were chosen by the team during sprint planning.

Daily Scrum or Stand-up

The *daily Scrum* or *stand-up* meetings are short, efficient meetings where the teams discuss their progress, highlight issues and blockers, and share their plans for the day. A scrum is usually held at the very beginning of the workday.

Increment

An *increment* is a goal achieved during a sprint. Each sprint should result in one or more increments. The correctness of the task or features achieved by the increment should be verified by testing.

Sprint Review

The *sprint review* is a meeting to inspect the sprint's outcome. These meetings are usually more than demos — the goal is to get feedback based on the Scrum team's results. As the aim is to reach the product goal, the stakeholders give opinions and suggestions about future adaptations. The outcome of these meetings can be changes or additions to the product backlog.

Sprint Retrospective

The goal of the *sprint retrospective* is to enhance quality and effectiveness — and not just in the current product. The retrospective should reveal hidden tensions within the team, any lack of information that slows down processes, outside dependencies that should be eased to lighten the burden of the team, etc. The team discusses what went well, what problems they faced, and what solutions were (or were not) found for those issues.

The outcome is a list of problems paired with possible solutions assigned to the responsible person.

Scrum Master

All of these meetings are hosted by the *Scrum master*. Each Scrum team needs one. They are responsible for facilitating Scrum processes and helping the team keep moving toward the product goal while issues are encountered during sprints. The Scrum master not only guides processes, but acts as a coach to ensure effective communication within the team.

Product Owner

The *product owner* is responsible for maximizing the value of the product created by the Scrum team. This role may vary across different organizations and teams. Even when delegating tasks, the product owner remains accountable for the outcomes.

Success requires organizational respect for the product owner’s decisions, which are reflected in the product backlog and the sprint review’s increment. The product owner is a single person, representing various stakeholders' needs in the product backlog. Anyone wishing to change the backlog must persuade the product owner to do so. The product owner defines and prioritizes the product vision and backlog and ensures the transparency, visibility, and understandability of the product backlog.

The Scrum master and product owner collaborate to drive project success. Their partnership helps the development team deliver high-value features efficiently and effectively.

Developers

The developers implement the requirements following the agreed sprint backlog. They can work independently, but there are several occasions where they might collaborate: for example, pair programming or reviewing another developer’s code.

Evaluating the Scrum Model

Scrum has become favored among software teams, but it too has pros and cons.

The Scrum Model benefits from:

Flexibility
Processes are flexible and iterative, allowing for changes based on feedback.

Customer involvement
Regular feedback from stakeholders ensures alignment with customer needs.

Improved quality
Frequent testing and reviews improve product quality.

Team collaboration
The model emphasizes teamwork and communication through daily stand-ups and regular meetings.

The Scrum Model suffers from:

Required discipline
Scrum calls for strict adherence to its practices, which can be challenging.

Scope creep
There is a risk that more and more customer requests will be added and hold up the release, if the product backlog is not managed well.

Role confusion
Standard roles such as Scrum master and product owner can be misunderstood or poorly implemented.

Resource intensivity
Daily meetings and frequent reviews can be time-consuming.

Kanban

Kanban is another popular agile framework that emphasizes work visualization, flow management, and process improvements. One of the big differences between Scrum and Kanban is that Kanban does not require fixed-length iterations. Thus, it makes continuous delivery more flexible and can balance different work priorities.

We’ll look at what teams need for a Kanban project in the following subsections.

Kanban Boards

A Kanban board is a visual tool that tracks the work’s flow through stages. The main and most important columns are “To Do,” “In Progress,” and “Done.” More columns can be added, but these three main columns have to be on the board. This visualization helps teams to catch bottlenecks and see the status of tasks in the blink of an eye.

Work in Progress Limit

A *work in progress (WIP) limit* helps to avoid multitasking and improves focus. With this limit in place, there is a point where no more tasks can be added to the “In Progress” column. In this way, the developers will not be overloaded by tasks and they can focus on the tasks on which they are currently working.

Team Members

Team members are responsible for completing tasks and establishing a smooth flow through the Kanban system. They collaborate to prioritize work and highlight and address any kind of issues that emerge.

Kanban Manager

The *Kanban manager* oversees the Kanban process, ensuring that the team follows Kanban principles and practices. This manager facilitates meetings, monitors workflows, and helps resolve any problem that may arise.

Evaluating the Kanban Model

The core of this model is simple. Let’s look at pros and cons.

The Kanban Model benefits from:

Visual workflow
Kanban boards provide clear visibility of work status and progress.

Flexibility
Lacking fixed iterations, the model supports continuous delivery with substantial adaptability.

Limits on tasks
The WIP limit helps prevent team members from getting overloaded and thus improves their

focus and efficiency.

Continuous improvement
The model encourages regular evaluation and process improvements.

The Kanban Model suffers from:

Lack of timeframes
Without set deadlines, time management is challenging.

Less structure
The model might lack the structure some teams need to stay organized.

Required discipline
Effective WIP limits and board management require careful attention.

Potential oversimplification
The task descriptions might oversimplify complex projects if not implemented thoughtfully.

DevOps

DevOps is a software development approach that integrates the development (Dev) and operations (Ops) teams to enhance collaboration, efficiency, and the speed of delivery. The primary goal of DevOps is to shorten the software development lifecycle and provide continuous delivery with high software quality. DevOps emphasizes automation, continuous integration and continuous delivery (CI/CD), and close collaboration between traditionally siloed teams.

Automation is crucial in DevOps for tasks such as code integration, testing, deployment, and infrastructure management. Automating repetitive tasks reduces errors, speeds up processes, and allows teams to focus on more strategic work.

Continuous monitoring and logging are essential to maintain system health and performance. DevOps practices include setting up comprehensive monitoring and logging systems to detect issues, analyze trends, and improve system reliability.

Evaluating the DevOps Model

Although DevOps is highly recommended for many types of modern software projects, pros and cons still have to be considered.

The DevOps Model benefits from:

Speed and efficiency
: The model accelerates development and deployment cycles through automation.

Improved collaboration
: The model bridges the gap between development and operations teams.

Continuous delivery
: The model ensures that software is always in a deployable state, leading to more frequent releases.

High reliability
: Automated testing and monitoring enhance reliability and reduce errors.

The DevOps Model suffers from:

Cultural shift
: The model requires a significant cultural change and buy-in across the organization.

Complexity
: Managing CI/CD pipelines and automated infrastructure can be complex.

Security risks
: Continuous deployment can introduce security vulnerabilities if not managed carefully.

Tool overload
: DevOps can be overwhelming due to the multitude of tools and technologies involved.

Guided Exercises

1. What does agility mean?

2. What is the Scrum definition, according to the Scrum Guide?

3. Why can Scrum perform better than the waterfall model regarding client feedback?

4. What are the positive aspects of DevOps?

Explorational Exercises

1. Why is the waterfall model not the best to use in a fast-changing environment?

2. What can happen when the role of the Scrum master is poorly implemented?

Summary

The relevance of project management in software development, especially in open source projects, lies in its ability to provide structure, enhance communication, define roles, manage risks, and ensure quality. These elements are crucial for the successful completion of projects and for fostering a collaborative and productive development environment.

In this lesson, we went through the waterfall model, agile methodologies, and DevOps. Knowledge of these models is essential to understand project management in software development. There is no golden right way to work — every project is different. In this lesson, you learned the roles, processes, and pros and cons of various approaches, which can help you in the future to understand and perhaps choose the right model for a project.

Answers to Guided Exercises

1. What does agility mean?

   Agility expresses the ability to react and move quickly, in both the physical and mental world.

2. What is the Scrum definition, according to the Scrum Guide?

   - A Product Owner orders the work for a complex problem into a Product Backlog.
   - The Scrum Team turns a selection of the work into an Increment of value during a Sprint.
   - The Scrum Team and its stakeholders inspect the results and adjust for the next Sprint.
   - Repeat

3. Why can Scrum perform better than the waterfall model regarding client feedback?

   Because feedback is collected during development, the final product can better meet client expectations.

4. What are the positive aspects of DevOps?

   Speed and efficiency — Improved collaboration — Continuous delivery — High reliability

Answers to Explorational Exercises

1. Why is the waterfall model not the best to use in a fast-changing environment?

   The waterfall model collects feedback only at the end of development. Thus, any requirement that was misunderstood by the developers has to be corrected at the very end. If the environment changes very quickly, this model should not be used because there is no provision for feedback in the middle of the development cycle.

2. What can happen when the role of the Scrum master is poorly implemented?

   A poorly implemented Scrum master role can hinder the team's ability to deliver high-quality products efficiently and can undermine the benefits of adopting Scrum. The team may lack proper guidance on Scrum practices and principles, leading to inconsistent or incorrect application of the framework.

   Without an effective Scrum master to remove obstacles, impediments can slow down progress and reduce team productivity. Miscommunication between team members and stakeholders can occur, resulting in misunderstandings and misaligned expectations. The team may experience decreased morale and motivation due to unresolved issues, lack of support, and ineffective facilitation of Scrum events. Inefficiencies may arise from poorly conducted meetings, lack of focus, and ineffective sprint planning and reviews.

055.2 Product Management / Release Management

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 055.2

Weight

2

Key knowledge areas

- Understanding common release types
- Understanding software versioning and reasons for major or minor releases
- Understanding the lifecycle of a software product, from its planning, development and release to its retirement
- Understanding the documentation for product versions

Partial list of the used files, terms and utilities

- Alpha and beta versions
- Release candidates
- Feature freeze
- Major and minor releases
- Semantic versioning
- Roadmaps and milestones
- Changelogs
- Long Term Support (LTS)
- End of Life (EOL)
- Backward compatibility

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	055 Project Management
Objective:	055.2 Product Management / Release Management
Lesson:	1 of 1

Introduction

Most software evolves in many ways over time. In fact, that’s one of the wonderful things about software: It’s easy to change and requires just a network transfer to update everyone who’s using the product. So developers add new features, fix bugs and security flaws, port the software to new hardware, and add interfaces to popular programs and services. As the software changes, new *versions* or *revisions* are released.

This lesson covers the logistics for planning and making changes to software, how teams juggle multiple versions, conventions for naming the versions, and other organizational aspects of development that are a subset of the actvities involved in *project management*. The logistics that apply specifically to the timing, naming, and control of releases is known as *release management*.

Characteristics of Releases

Releases vary in terms of stability, backward compatibility, and support. We’ll examine each of those concepts in the following sections.

Stable and Unstable Releases

Stability is a key trait users expect from software. The first question many ask, upon hearing of a new release, is “How stable is it?” In other words, will it work reliably, or will it crash, corrupt data, or produce incorrect results?

Stability can be measured on a spectrum, and many people are happy to use software even when there are still some bugs. But development teams tend to keep things simple, and talk in binary fashion about *stable* and *unstable* releases.

Stability refers both to the bugginess of the software and its likelihood to change in ways visible to outsiders. Developers might consider a release of a software library unstable because they have changed the functions that programmers call (that is, because programmers might have to rewrite their software for the next release). Or software might be unstable from a user’s point of view because developers have removed a feature.

Is there any reason to release an unstable version? Yes: They are valuable for letting users try out new features early in the project and test for bugs.

Most projects release very early revisions of software for key clients to try out; these are called *alpha* releases. No one should use these releases for real work — they are only for testing. In fact, testers should run alpha releases on computers that aren’t doing any important work, because bugs in those releases could actually damage data stored on the computer.

When the software is close to being considered stable, the project usually release another test revision called a *beta* release. These should still not be used for production work, only for testing.

For both alpha and beta releases, the developers have a process for reporting bugs and tracking the progress toward fixing bugs.

Some projects include another stage between the beta and stable releases, when developers have fixed all the bugs they can and think that the product is done. They call the version a *release candidate* and show it to key customers or stakeholders so that these stakeholders can plan how to use new features.

Finally, some projects put in features that are not quite stable, because important users have requested them. The developers intend for users to try these features and say whether they like them, before developers finalize the interfaces and invest the effort to stabilize the features.

Backward Compatibility

Most projects seek *backward compatibility*, meaning that they try not to remove features or functions that existed in older versions. Compatibility can exist on several levels. For instance, if the developers maintain backward compatibility for the application programming interface (API), they promise that old source code can continue to work, but it might have to be recompiled. If hardware vendors or operating system developers maintain backward compatibility for the application binary interface (ABI), they promise that in addition, old executable files can run on new versions of the hardware.

We'll look later at the ways projects handle a *lack* of backward compatibility, when they decide that the old interface really doesn't work for the new features or environments they need to support.

Support and End of Life (EOL)

Software would ideally get better and better as developers discover and fix its problems. But over time, features can stop working because of changes in the software's environment. Also, new bugs are discovered that were hidden before.

New versions often combine security and bug fixes with new features. You might not want the new features (in fact, you might prefer some old feature that the developers removed in order to make way for new ones), but people usually upgrade to the new version to get the security fixes that will protect them against malicious attacks.

Some people actually keep using old versions of software, refusing to upgrade. This is normally because the new version breaks something that was working in their environment. When companies charge money for upgrades, some customers refuse to upgrade because they don't want to pay the extra money, but paying for upgrades is rare in open source.

Developers accommodate the users of old versions, if possible, by fixing bugs in the old versions without adding the features that break the software. Naturally, the developers can't do this forever because it drains time and energy from new work. There will come a time when they refuse to fix an old version and tell the users either to upgrade or to make their own fixes.

Fixing bugs and security flaws is called *support* for the software. This is different from the support offered by helpdesks and other people who guide users in understanding the software. So far as release management goes, a *supported release* is one where the developers promise to fix bugs, whereas an *unsupported release* is one where they do not.

Note that the notion of "support" applies mostly to software created by companies or large organizations. Smaller open source projects, depending heavily on volunteers, often promise just to do the best they can to fix bugs and put out new versions. They don't see a need to fix old versions. Because the code is open source, users who don't want to upgrade can pay someone to fix an old version.

When support is in place, developers publish a schedule indicating how long they'll support each version. The date where support ends is called the software's *end of life* (EOL). Users can keep the old version up to EOL and expect bugs to be fixed, but after EOL they have to upgrade or take their chances.

Major projects, such as the Debian distribution of GNU/Linux, offer *long term support* (LTS) versions. That simply means that the developers will keep fixing bugs for a certain number of years. Customers who value stability highly might be afraid to install versions of software that have lots of changes, in case the changes break processes that the customers rely on. Such customers, notably large institutions, like the security of LTS versions.

Software Versioning: Major, Minor, and Patches

We've seen that versioning is complicated: Some versions fix bugs, while others add features, and the versions can vary in stability. Developers try to indicate through labels how big a change to the software is in each version. The conventions adopted by nearly all projects for labeling versions are called *semantic versioning*.

Let's take the early history of the Linux kernel as an example. Linus Torvalds labeled the first stable release 1.0. As he improved the kernel, he labeled subsequent versions 1.1, 1.2, and so forth. The initial 1 was the *major version* and the numbers after the period denoted the *minor version*. You could assume that version 1.2 had more features and offered more than version 1.1 did.

But there were also innumerable small releases, sometimes just to fix a few bugs. To show that the changes had a minor impact on the use of the kernel, Torvalds included a third number called the *patch*. Thus, 1.0 was upgraded to 1.0.1, then 1.0.2, and so on.

Patch numbers start over at 0 when a new minor version is released, and minor version numbers start at 0 when a new major version is released.

Developers lump together versions using an "x" to indicate that they're talking about multiple versions, such as 1.x for all versions under major version 1.

But what's the difference between a change that leads to a new minor version and a change that deserves to trigger a new major version? Usually, years pass before a new major version is released, and it has to reflect a very significant upgrade.

Generally, developers try to keep backward compatibility as versions change. If backward compatibility cannot be preserved, developers should increment the major version.

Sometimes you see a version number less than zero, usually 0.9. The leading zero indicates an early version of the software that is unstable and unready for production use. There is no guarantee of backward compatibility until the developers release versions starting with 1.

Alpha versions are usually indicated by appending “a” or “alpha” after a release number, such as 3.6alpha. Similarly, beta versions append “b” or “beta” after the release number.

The Software Product Lifecycle

Don’t think the life of a developer is easy. Everybody wants something from them. I want this bug in screen layout fixed A.S.A.P., whereas someone else says that the bug in filenaming takes precedent. Users clamor for new features, and when different developers work on different features in parallel, they find that one’s changes may trample those made by another.

Project management, and the subset of these tasks called release management, deal with these issues. Usually, a senior project member takes on responsibility for this management job.

Like people, software versions go through a lifecycle. Each one begins with a discussion of new features and other changes needed, goes through development and test phases, and is rolled out in a planned manner.

Planning and Roadmaps

Developers try to lay out, far in advance, what changes they want to make to a product. In commercial firms, they talk to marketing people, who filter and summarize what their customers tell them. Open source projects depend more on ideas submitted by developers and users, which are captured in a database called an *issue tracker*. If you need a bug fixed or want a new feature, you fill out an *issue*. Developers then prioritize the changes and decide who will handle each one.

So how are changes chosen and prioritized? It can be a messy process. But good project managers on open source projects encourage broad input while ensuring that decisions get made. Some projects even hold conferences where participants hash out the priorities.

A published *roadmap* lays out the resulting plans for enhancements and changes. It may extend many releases and many years into the future. Each step is called a *milestone*, and might or might not be associated with a target date.

Release Scheduling

There are two basic ways to schedule releases: time-driven and feature-driven. A project can promise a release at regular intervals — every six months, for instance — and include whatever is finished at that point. Alternatively, the project can promise to include certain features in a release, and let the developers take as long as they need to finish those features.

As the time for the release approaches, the release manager determines dates for the alpha, beta, and stable releases. Team members regularly review bug reports and try to plan their work so that they meet these milestones.

In order to finish a release, a project has to stop accepting ideas for new features and focus on making the existing features work right. This moment is called a *feature freeze*.

Documentation for Product Versions

Roadmaps, as mentioned earlier, explain the features that developers plan to include in each release. The release is accompanied by a list of changes called a *changelog*. Generally, the changelog lists new features, changes to existing features, features that were removed, features that the developers intend to remove in the future (known as *deprecated* features), and bug fixes.

Thus, changelogs can get quite long and detailed. Users need to pay special attention to features that were removed or are deprecated, because the users might have to change their programs or way of using the product. Obviously, developers try to remove only those features that nobody needs any more.

Each release should change product documentation to match changes in the product. This task can be time-consuming, and it’s easy for developers to miss a change or lag at producing documentation.

Guided Exercises

1. What distinguishes a stable release from an unstable one?

2. Would you expect a feature change between a release named 2.6.14 and a release named 2.6.15?

3. Would you expect a feature change between a release named 2.6.0beta and a release named 2.6.0?

4. Why would you expect release 1.0 not to be backwardly compatible with release 0.9?

5. If you discover a security flaw in a version after a feature freeze but before its release, can you get the flaw fixed?

Explorational Exercises

1. Suppose you want to keep using a version of open source software after EOL, because it has a feature you need. What can you do to keep it usable?

2. What are some criteria that allow one bug fix or feature request to be chosen over others?

Summary

This lesson described the main characteristics that distinguish releases: stability, backward compatibility, and support. We discussed the meaning of version names and numbers, and key aspects of release management, including documentation.

Answers to Guided Exercises

1. What distinguishes a stable release from an unstable one?

   Releases are considered stable in two ways: They work well without crashing or producing incorrect results, and the interface presented to users or programmers is expected to be backward compatible with previous versions.

2. Would you expect a feature change between a release named 2.6.14 and a release named 2.6.15?

   No. The third number in semantic versioning indicates a patch, which was made to fix a bug or carry out some other minor task such as reformatting. A feature change should lead to a minor or major release.

3. Would you expect a feature change between a release named 2.6.0beta and a release named 2.6.0?

   No. The beta version is a test version that contains all the features to be included in the final release.

4. Why would you expect release 1.0 not to be backwardly compatible with release 0.9?

   The number 0.9 explicitly warns potential users that the software is still being designed and could very likely have a different interface when it is stabilized as version 1.0.

5. If you discover a security flaw in a version after a feature freeze but before its release, can you get the flaw fixed?

   Most certainly. The period between a feature freeze and the release is meant as a time to discover and fix flaws, including security flaws.

Answers to Explorational Exercises

1. Suppose you want to keep using a version of open source software after EOL, because it has a feature you need. What can you do to keep it usable?

   After EOL, the project developers have no commitment to fix bugs, including security flaws. Therefore, you should follow the project's bug tracker and mailing lists assiduously to find out what bugs turn up. Because the code is available, you can and should fix bugs that are in your version. In theory, you could even incorporate new features into your version. That would effectively make your version a fork of the original.

2. What are some criteria that allow one bug fix or feature request to be chosen over others?

   For bug fixes, criteria include severity (which is assigned to the bug after it enters the bug tracker) and the number of users affected by it. For a feature, criteria include the number of users who want the feature, the difficulty of coding it, and its potential impact on other parts of the program.

055.3 Community Management

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 055.3

Weight

2

Key knowledge areas

- Understanding roles in open source projects
- Understanding common tasks in open source projects
- Understanding the various kinds of contributions to open source projects
- Understanding the various kinds of contributors to open source projects
- Understanding the role of organizations in maintaining open source projects
- Understanding the transfer of rights from individuals to an organization maintaining a project
- Understanding rules and policies in open source projects
- Understanding attribution and transparency on contributions
- Understanding aspects of diversity, equity, inclusivity and non-discrimination

Partial list of the used files, terms and utilities

- Software development
- Documentation
- Designs and artwork
- User support
- Developers

- Release managers
- Users
- Project leads and benevolent dictators
- Individuals and corporations
- Enthusiasts and professionals
- Core team members and occasional contributors
- Code and documentation contributions
- Bug reporting
- Forks
- Foundations and sponsors
- Contribution agreements
- Developer Certificates of Origin (DCO)
- Coding guidelines
- Codes of conduct

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	055 Project Management
Objective:	055.3 Community Management
Lesson:	1 of 1

Introduction

A key reason to create an open source project is to draw contributions from many different people. Open source makes it easier to respond to the different needs and interests of the project's users and to benefit from their varied skills. Therefore, a diverse community is central to the success of the project. The community is where creative forces come together to make your project succeed.

This lesson explains the basic elements of free software and open source communities and how people in them work together. Of course, because communities are made up of different people, and because projects have different goals and requirements, each community is unique.

Roles in Open Source Projects

The main output from most open source projects is software, so at the very least, such projects require programmers, software designers or architects, testers, release managers, and other experts in code development. Documentation is usually a part of such projects too, so the community also needs authors, editors, and reviewers.

Other projects might not produce code, but other “deliverables” such as a book, an art or music project, or a policy report. Wikipedia is a well-known example of collaboration on an open source project that focuses on text documents and media. These projects need experts in the design, production, and delivery of the deliverable.

Communities also benefit from many other general skills, such as marketing and evangelism, administration, legal advice, artistic talent to produce logos or diagrams in documentation, and skills at maintaining the project’s web site and other communication tools. Open source projects might organize physical get-togethers and even conferences, in which case they need organizers to bring those events to life.

The proceeding paragraphs give you an idea of the types of skills a community looks for. These general roles can be further subdivided into various focused tasks. One simple example of such a task is for some writers to edit the work of other writers.

Among programmers, a range of experience exists. A healthy project always wants some *veteran* programmers (or *senior* programmers) who know its code and coding standards well, along with *newbies* who provide new ideas and simple help such as bug fixes. Some newbies will hopefully evolve into the next generation of veterans.

Code quality requires careful control over what gets into the central repository shared with users. Therefore, some veterans get *committer* status. They are responsible for checking contributions for quality, usefulness, and conformance to coding standards. They have the priviliges to accept the proposed code into the trusted repository. Other contributors submit code to the committers for review.

Many committers and other veterans also *mentor* new contributors to teach them the standards and practices needed to get their code accepted.

Some committers fail to appreciate the gift that a contributor is offering. If the contribution falls short of quality or coding standards, the committer might simply reject it. But the rejection discourages the would-be contributor and eliminates a valuable educational opportunity. Remember that good committers are also mentors. So they give the contributor hints about how to bring the code up to standards and work with the contributor over time. If the contribution is truly unusable, the contributor should at least be thanked and encouraged to work on something else for the project. It’s important to help people discover and refine their skills.

When a company starts an open source project, it sometimes appoints committers from its own staff to ensure that it can control the direction and quality of the project. But often, companies allow non-employees who show skill and dedication to become committers.

Project leads make critical decisions such as what new features to support. These project leads often have non-coding decisions to make as well, such as how to promote the project, resolve major disagreements, and get funding. Project leads and committers work closely with *release managers* to decide when a code base is stable and ready to share with the users.

Some projects have a single project lead, often referred to as *benevolent dictator*. This is often a person who started the project (Linus Torvalds being a well-known example in the case of Linux) and who has great authority and even charisma. However, most projects prefer to set up a small committee of project leads instead of a single benevolent dictator. Even the Linux project has moved to a committee approach.

Projects might also ask particular contributors to provide user support on the project’s forums. But healthy projects should have many knowledgeable users who support each other.

We’ll finish this section with a very important role: the *community manager*. This is someone responsible for maintaining open and constructive communication and making sure the community progresses toward its goal. The community manager knows how to onboard and motivate contributors, resolve arguments, protect community members against abuse, and perform other tasks to maintain the community’s health.

Common Tasks in Open Source Projects

In many ways, open source projects involve the same tasks as traditional projects carried out within a single organization. For instance, all code requires testing. But in some ways, open source projects differ from proprietary ones, where only a limited set of people are allowed to change the code and where the source code is often hidden from the public.

For instance, corporations usually assign trained quality assurance (QA) staff to test code. But many open source projects just ask their users to test each release. (Proprietary projects involve their users in testing too, in addition to QA.)

Another example of differences: A proprietary project usually assigns paid developers to specific tasks and tells them how much of their time each week to spend on the tasks. Some open source projects benefit from contributors who are paid by their employers, or sometime even employed by the project itself, and who are assigned tasks in the same way as proprietary developers. But in most healthy projects, a lot of contributions come from volunteers who just do the tasks as time permits. Because the project doesn’t rely on all volunteers to complete their projects on schedule, an open source release might be delayed until the developers feel that the features are ready, or might be released at fixed times with whatever features are ready.

Communications are critical to both proprietary and open source projects, but are often conducted differently. Proprietary projects often gather a team in one office and hold regularly scheduled stand-up meetings under a development methodology such as SCRUM. Because open source projects are geographically distributed, such methodologies are rare. Instead, communication takes place online and asynchronously.

In short, free software and open source projects often achieve the same goals as proprietary projects, but in different ways.

Bug reporting is a critical part of software development that gives rise to its own set of roles. Someone has to monitor the bug database and assign priorities. If a bug is critical (and particularly if it can weaken the software's security), it should be assigned to a competent maintainer to fix.

Project leads and community managers should look over participation in the project and see where there are gaps. Are too few people willing to test code? Is documentation missing? Are too many veterans or senior project members leaving without being replaced? Project leads and community managers can focus on recruiting people to fill needed roles.

Project leads and community managers on large projects often gather metrics as well, to learn important things such as whether important bugs are getting fixed in a timely manner.

Kinds of Open Source Contributions

As explained in an earlier section, people who put code, documentation, or other items in the central repository are called *contributors*. Project members who approve contributions and accept them into the repository are called *committers*. Some members take on other common roles for software development.

Although the term *contributor* is usually reserved for someone who develops code or another part of the project's official offering, anyone who participates to the project's success is contributing in some way and should be thanked for doing so. These more informal contributions include reporting a bug, answering a question on a forum, donating or raising funds, and promoting the project to outsiders.

Open source projects, like proprietary ones, ask users what they want in terms of features, performance improvements, or other changes to the project. These users are making important contributions by voicing their opinions.

Kinds of Open Source Contributors

Many communities draw contributions not only from individuals, but from corporations who pay their staff to contribute to a project that the corporation thinks is important to its business.

Sometimes, having paid contributors right next to volunteers can create tensions. Volunteers might feel that their work is being exploited, or might be afraid that paid contributors are trying to bend the direction of the project to favor their employers' interests. A community manager and the project leads must guarantee that all accepted contributions provide benefits to the whole project and its users. Volunteers must also receive motivation to contribute for their own personal reasons, whether out of loyalty to the community or to meet their needs.

Some people contribute to a project regularly and take on long-term responsibilites; these are called *core team members*. Healthy communities also have occasional contributors who report bugs or post advice on forums, but are not expected to assume responsibility for the project.

Similarly, some contributors will be professionals in their field — whether or not a company pays them to work on the project — whereas others are amateurs, often called *enthusiasts*. But you don’t have to be a professional to take on an important role. You might even become a core team member or project lead.

The Role of Organizations in Open Source Projects

Although many open source projects are created by zealous individuals or small groups of volunteers, they usually seek organizational support when the projects get large. Organizational support can take many forms. In general, organizations make these contributions because an open source project can meet their business needs more cheaply and more reliably than reinventing the same features in proprietary code. The guarantees provided by a free or open source license are valued by many organizations.

Some idealists distrust corporations or other large organizations and would prefer to keep open source projects entirely free from their influence. Over the years, this rather simplistic attitude has become less common. Most open source advocates believe that corporations and other organizations can provide crucial underpinnings to open source projects.

Many open source projects start within an organization, and might even be based on proprietary code that the company decides to open up. To make money, the company may decide to keep firm control over the project and to break it into open and proprietary parts: This is called the *open core* model. Many project founders start as open source but form a company around it, which might or might not use the open core model.

Other projects try to ensure that no single company controls their direction. Maintaining independence usually requires signing up several organizational supporters, so that no one is strong enough to make a dictatorial decision.

How do organizations support open source? As mentioned earlier, many put paid staff onto projects. Additionally, they might hire highly productive individuals who have contributed to the project as volunteers and have developed expertise in the project. This path to employment is a valuable way for students and other volunteer contributors to advance their careers.

Companies desiring extensions that don't interest other users should not pressure the open source project to add the extra features, but should pay their staff to write the features without contributing them back to the project.

An interesting example of the possible tension created by corporate needs is shown by the famous Android operating system, based on Linux and developed by Google to drive its mobile devices. Some changes to Linux made by Google are contributed back to the Linux community, whereas other changes appear only in Android. The Linux developers sometimes reject changes submitted to them by Google, just as all projects choose what contributions to incorporate.

There are many reasons for a company, or a group of developers, to make a separate version of your code. Ideally, they can meet their needs by adding an optional library or sequence of code that can be excluded during compilation. But sometimes a group feels a major need that is incompatible with the direction chosen by project leaders. When a separate version is made, it's called a *fork*.

Forks used to be considered failures of collaboration, but nowadays forks are much more accepted. Usually, people who make a fork set up a new repository and start a new project. Some people work both on the original project and the fork.

(Note that GitHub uses the word "fork" for a very different phenomenon: making a clone or copy of project code to work in separately.)

In addition to code, many companies contribute financially. They may fund efforts such as marketing and conferences. They can join the board and offer expert advice on the project's direction and strategy.

An example of the importance of such "soft support" comes from the historic Apache web server. The project took a big leap forward early in its existence when IBM showed interest. IBM lawyers showed the project how to create legal safeguards, and a meeting paid for by IBM helped the Apache leadership create a robust organization.

To maintain independence, many open source projects form a nonprofit foundation or join an existing foundation. Famous examples of foundations that guide open source projects include the Linux Foundation, the Apache Foundation, and the Eclipse Foundation.

A foundation's support is valuable because it can provide the logistics that most software developers don't want to deal with: legal support such as trademarking, indemnification, and licensing; help with raising money; infrastructure such as bug databases and web sites; and so on.

Transfer of Rights

Just as with books, music, and other creative efforts, software involves a complicated relationship between the developers, the organizations that distribute their contributions, and the general public. Essentially, contributors must take steps to formalize the right of an open source project to use and distribute their code.

Thus, many open source organizations ask contributors to sign a contributor license agreement (CLA). Sometimes, the contributor just gives the code to the project. The project owns the code and all rights to it, just as a proprietary company owns the code that it paid its staff to develop.

Other CLAs leave some rights in the hands of the individual contributors. Contributors might like this flexibility because they can go on to contribute the same code to another project or build their own business around it.

To harmonize the different contributions from different people, the license assigned to the code is crucial. The Linux kernel is one of the projects that leaves ownership in the hands of the contributors, so that by now many thousands of people own rights to the Linux code. But all contributors to the core code put it under the GNU General Public License (version 2). Thus, Linux is free for all to use, alter, and redistribute.

Using a single license for all code in a project is the simplest way to guarantee that no encumbrances hinder the code from being distributed and used. But sometimes a project allows different parts of code to be contributed under different licenses, usually because the project wants to take advantage of some pre-existing code that's already released under a different license. Licensing experts should make sure that the licenses are compatible.

Rules and Policies

Many online communities have a reputation for unrestricted speech (a cavalier idea that “anything goes”) leading to verbal abuse and bickering. Nowadays, most open source communities are battling that tendency, which takes hold of people all too easily. In contrast, modern communities want people to be constructive, civil, respectful, and inclusive of everyone: all genders, ethnic groups, personality types, and so forth.

Expectations of group members are usually specified explicitly in a *code of conduct* that explains how to engage with other people on the project, both online and in person. To be effective, this code of conduct has to be enforced by the community manager and project leaders.

Sometimes, a person who flagrantly mocks or denigrates a fellow community member is expelled permanently or temporarily. At other times, the community manager or a colleague simply announces publicly that the behavior violates the code of conduct and might talk to the offender outside the forum. Often, the offender previously felt frustration, lack of attention, or burn-out, and community members can help the offender find better ways to express themselves.

In addition to social behavior, communities establish quality standards. The most formal consist of *coding guidelines*, which try to ensure that all code looks similar. The guidelines may remind developers of good practices such as using brackets around blocks of code, or may specify details such as how to name variables and what kinds of indentation to use.

Open source communities also have rules around releases of code or other deliverables. Some projects establish fixed times for releases; for instance, Ubuntu promises a new long term support (LTS) version every two years along with interim releases on a more frequent basis. Other projects maintain a list of new features and bug fixes they want in the upcoming release and approve the release when everything is checked off the list. But unlike most businesses, open source communities don't usually establish deadlines for participants, because one cannot ask too much of volunteers.

Attribution and Transparency

In addition to the contributor license agreement mentioned earlier, some projects ask contributors to sign a *developer certificate of origin* (DCO). In the DCO, the contributor promises that they have the legal right to donate the code.

Why is the DCO important? Imagine this scenario: If a programmer takes code from a proprietary product produced by their employer and contributes it to an open source project, the programmer violates the employer's license and puts the open source project at legal risk.

The DCO is supposed to ensure that the contributor actually wrote the code, or obtained it legally. The open source project depends on the honesty of the contributor who fills out the certificate.

Diversity, Equity, Inclusivity, and Non-Discrimination

Social scientists claim that projects and companies benefit by having many people with different genders, races, economic backgrounds, nationalities, and abilities. All organizations have a natural human tendency to bond with other people like their current members, so a community that values diversity and equity has to consciously train its members to be more open to people who aren't like them. The movement to carry out this ideal is called diversity, equity, and inclusion (DEI).

The code of conduct is the starting point for DEI. It should explicitly welcome people of different genders, races, etc. Every violation of the code of conduct must be handled promptly with unambiguous disapproval. This is because some minorities have suffered from exclusion and negative comments their whole lives, and a single bad interaction in your forum could make them decide they have no reason to participate anymore. A speedy response to aggressions can reassure them that the community backs them.

But DEI goes much further. The community should reach out to groups who need more representation. For instance, if you are developing a job search app, you should make sure it shows jobs for low-income people as well as upper-class and middle-class people. You should also advertise the app among low-income communities and recruit members from those communities to test it.

Your community might benefit from finding organizations that train people from marginalized communities, and from which you can recruit new members of your team. Many such organizations are established in local areas, and aren't well-known nationally or internationally.

Understanding the needs of marginalized communities is key. Is your web site accessible to the sight-impaired? Do you translate your documentation into languages with which the target community is familiar? Perhaps you should create specific forums in other languages.

Most communication in open source communities is asynchronous and online. But if you have meetings or synchronous chat sessions, think about where people are geographically and find ways to include everybody. For instance, when people from many countries and regions are communicating in English, try to keep the vocabulary and grammar simple.

Finally, if you have members of some minority on your team, make sure to listen to them. One well-known symptom of exclusion is discounting what minorities say or just underappreciating its importance.

On the other hand, don't burden minority members with the need to explain their communities' needs — everybody should be tasked with that research. The minority members might do valuable outreach for you, but don't pressure them to do so. Each person should have the equal opportunity to participate the way they want, without having to be a token minority.

Guided Exercises

1. Why don’t all contributors give their code to the project and relinquish all rights to the code?

2. If someone wants to help the project but can’t program, what are some ways they can help?

3. Why do many open source projects join a foundation?

Explorational Exercises

1. You are a committer on your project. Someone submits code that was taken from another project (but the contributor has the rights to it). The code is formatted completely differently from the rest of your code. What do you do?

2. You created a proprietary software product and want to contribute parts of it to an open source project. Under what circumstances can you continue to offer your proprietary product?

3. Two people on your mailing list start to argue over a feature in your code. The argument gradually gets more heated until one person calls the other an idiot. How can you handle the situation?

Summary

In this lesson, you learned what it’s like to be part of a community and how communities remain productive. You learned different types of contributions, contributors, and roles. You learned how communities control the right to use contributions. You learned about rules in a community and how they protect everyone, including people of different races and genders, from verbal abuse.

Answers to Guided Exercises

1. Why don’t all contributors give their code to the project and relinquish all rights to the code?

   The contributor might want to contribute the same code to a different project or release a product based on the code, and therefore wants to keep some rights.

2. If someone wants to help the project but can’t program, what are some ways they can help?

   There are many roles for contributors besides coding. A few such roles include documentation, testing and bug reporting, community management, participating in forms, promoting the project, and doing artwork.

3. Why do many open source projects join a foundation?

   A foundation handles many legal, financial, and other tasks that the project’s community might have difficulty doing.

Answers to Explorational Exercises

1. You are a committer on your project. Someone submits code that was taken from another project (but the contributor has the rights to it). The code is formatted completely differently from the rest of your code. What do you do?

   Your project's coding standards should clearly explain how to format the code. Thank the contributor, point them to the standards, and ask them to reformat the code. Sometimes, automated tools exist to reformat code as you want it. If the contributor doesn't have time to do the reformatting, look for a junior member of your team who can do that job.

2. You created a proprietary software product and want to contribute parts of it to an open source project. Under what circumstances can you continue to offer your proprietary product?

   The answer depends on the contributor license agreement. If the CLA requires you to hand the code over to the project, granting them all rights, you might not be able to continue offering your proprietary product. However, if they let you keep your rights to the code, you can release it under any license you like in your proprietary product.

3. Two people on your mailing list start to argue over a feature in your code. The argument gradually gets more heated until one person calls the other an idiot. How can you handle the situation?

   Anyone who notices the escalation and the abusive comment should react as quickly as they can. The community manager is ultimately responsible for repairing the damage. The person who intervenes should post a comment to the whole list saying that the behavior violates the project's code of conduct (which hopefully rules out such behavior.) The person intervening can also reach out to the people on each side separately to make sure they are satisfied by the resolution of the argument and understand how to discuss disagreements constructively.

Topic 056: Collaboration and Communication

056.1 Development Tools

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 056.1

Weight

2

Key knowledge areas

- Understanding common software development tools
- Understanding common deployment environments
- Understanding common types of software testing
- Understanding the concepts of Continuous Integration and Continuous Delivery (CI/CD)

Partial list of the used files, terms and utilities

- Integrated Development Environments (IDEs)
- Linters
- Compilers
- Debuggers
- Unit testing
- Integration testing
- Acceptance testing
- Performance testing
- Smoke testing
- Regression testing

- Production, staging and development systems
- Local development systems
- Remote development systems
- CI/CD pipelines

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	056 Collaboration and Communication
Objective:	056.1 Development Tools
Lesson:	1 of 1

Introduction

There are thousands of software development tools, both open source and proprietary. And why not? Programmers love to develop tools for themselves and their colleagues; it's only natural that they'd invest a lot of time trying to find a tool that works better, removes some annoyance from their workflow, or makes deployment easier.

This lesson focuses on the development *process* and explains how various types of development tools fit into it. Few specific tools will be named, because any one of them could possibly be flagged as deprecated or obsolete, and be replaced by a new favorite by the time you read this lesson.

Goals of Development

Programming tools juggle multiple goals that sometimes conflict with one another. Here are some sample development goals:

- Produce robust, accurate programs.
- Produce programs that run fast.

- Produce programs that scale well.
- Produce programs that are highly adaptable to different types of users, different devices (such as laptops, cell phones, and tablets), and different environments.
- Speed up the development process.
- Speed up deployment of newly developed features or bug fixes.
- Reduce tedious work, as well as the number of clerical programming errors that slip through to the test stages of the program.
- Support legacy code and devices that the organization has trouble replacing.
- Work together with other familiar tools.
- Allow for easy roll-back in case of errors or changes in plans.

These goals, and others, lead to the processes described in this lesson and the resulting development tools.

General Development Processes

This section contrasts two general models of historical importance: the *waterfall model* (introduced in an earlier lesson) and *continuous integration/continuous delivery* (CI/CD).

Waterfall Model

Although the waterfall model is still widely used, especially by large organizations, it has fallen out of favor among many developers. This model was popular from the 1950s through the 1970s. Elements of the model can still be widely found today, such as formal definitions of requirements and testing programs just before their release (*quality assurance*).

Even at the time that the term “waterfall” was coined for this model, it had become widely discredited. Problems included:

- Changing requirements was difficult once development started.
- Requirements and designs were often misunderstood, because plain-language text can be ambiguous. This problem led to products that didn’t meet the intended requirements, and took months to fix.
- The process was slow. A new release might be achieved only once a year, or even less often.
- Bugs caused by the interaction of different modules were hard to catch until late in the process, leading to further delays.
- The process put up barriers between different parts of the organization, which was bad for the

quality of the product as well as organizational *esprit de corps*.

Principles of Continuous Integration/Continuous Delivery (CI/CD)

The waterfall model was challenged in the 1970s by a series of movements that led to the most celebrated (if not universally used) model today: CI/CD. Stages in the adoption of the CI/CD practices include the *Agile Manifesto*, released in 2001, SCRUM, and DevOps. The new model is founded on the principles of tight communications among team members, involvement by the end user or customer, rapid roll-out of bug fixes and new features, and rigorous ongoing testing to preserve quality in a fast-moving — sometimes even chaotic — environment.

CI/CD automates as many steps as possible in the stages that range from writing code to deploying the completed program to end users. CI/CD requires defining each step formally and replacing a human action (such as installing software by hand) with a procedure run by a program. Thus, CI/CD is part of a movement known by the phrase “everything as code” and “infrastructure as code.”

Furthermore, once a team has incorporated its processes into code, these processes can be fixed, upgraded, and recorded historically just like code. Anything written by the team, whether programs, automated procedures, or documentation, should be stored in a version control system, which is described in an upcoming lesson.

When several procedures are automated, they can run in sequence. Thus, teams often talk about a *CI/CD pipeline*, which means that one successful step in the pipeline automatically triggers one or more subsequent processes. A pipeline must be monitored in an automated fashion so that any failure along the way causes the pipeline to stop and notify team members of the failure.

Although the following sections discuss CI and CD separately, they tend to be combined, and the dividing line between them is blurry.

Continuous Integration (CI)

Continuous integration refers to the fast incorporation of small changes into the code. The central repository used to create the product for end-users is known as the *core repository* (or *core repo*). Each programmer creates a personal workspace (often called a *sandbox*) on their computer and pulls from the core repository the files they need to fix or upgrade.

The programmer could use a personal laptop or desktop (known as a *local development system*), downloading relevant parts of the core repository and then uploading changes after local testing. Alternatively, the programmer could take advantage of the popular trend called “cloud computing” and work on a shared system run by their organization, a *remote development system*.

The programmer generally checks for errors using a *debugger*, a separate program that runs the programmer’s work in a controlled environment. We’ll look later at debuggers and other tools for catching errors.

To catch errors as early in the development process as possible, the programmer also runs tests on the code before uploading it to the core repository. These are called *unit tests* because each focuses on one tiny element of the code: For instance, did a function increment a counter as it was supposed to?

The last stage in CI is checking the programmer’s changes into the core repository. At this stage, tools run *integration tests* to make sure the programmer hasn’t broken anything in the project as a whole.

No matter how far automation has come, some expert member of the team should intervene at the point of integration to make sure the change is desired by the team. Automated tests can determine that nothing has broken, and even whether the change creates the desired effect in the program. But these tests can’t check for all the principles considered important by the team.

Therefore, before proceeding to deployment, a team member should check security, adherence to coding standards, proper documentation, and other principles. (Not surprisingly, tools exist for some of these tasks too; we’ll look at them later in this lesson.)

A lot of testing takes place during CI, and it has to happen both fast and reliably to support the pace of modern development. Therefore, modern tools for integrating code and running tests are automated. A programmer should be able to run a whole suite of unit tests through a single command or a click of a button.

Generally, a partial or full integration test runs whenever the programmer uploads code to the core repository. Any errors caught by the tests can be reported quickly to the programmer.

Continuous Delivery (CD)

As explained in the previous section, CI consists of automated procedures that lead up to a new version of the program in the core directory. *Continuous delivery* refers to anything that happens after that stage to bring the new version out into the field where end users can benefit from it. The D in CD is sometimes expanded to “development” or “deployment” in addition to “delivery.”

The main tasks of CD are to test the code thoroughly and to upload it to the users' computers or an application repository.

The CD phase runs a battery of tests to provide maximum assurance that the product works well. A larger set of integration tests might be run, along with a number of other tests to determine impacts on the users, performance, and security. A later section in this lesson describes some of these tests.

Testing should be done on different systems from the production systems that serve users. Not only could a catastrophic flaw bring down a production system; it might corrupt user data. Furthermore, tests compete for system time and degrade performance for users.

So for testing, it’s best to set up a complete computing environment that mirrors your production environment, with similar hardware and software. The intermediate environment where testing runs before deployment is often called a *staging* environment.

Of course, it wouldn’t be practical to make the test environment the same size as the production environment, but the essential elements of the production environment (such as databases) should be included.

One requirement of CD automation is to distinguish between the test and production environments. All the code installations and processes should be tailored to the target environment.

CD offers the most potential for rapid development when the code is a service running on the organization’s own systems. If you’re a retail site, for instance, offering an interactive web page, you have access to all your web servers. You can update them several times a day, if you want, and roll back changes quickly if they turn out to produce problems for users.

Common Software Development Tools

These sections present common types of tools used by programming teams at three levels: code generation, testing, and deployment.

Compilers

A fundamental programming tool is the *compiler*, which turns very sophisticated, high-level programming languages into the instructions running on the computer processor.

Computers run *machine code*, which consists of strings of bits (ones and zeros) that the processor turns into instructions and data: loading a value from memory into a register, adding the values of two registers, etc. Processors are distinguished by the different formats and sets of instructions they have, so they have different machine code as well. Vendors try to invent new processors that use the same machine code in order to allow customers to run their old programs on the new processors. When vendors do so, we speak of *families* of processors.

The next stage in the early years of programming was *assembly language*, which provided human-readable terms, such as ADD, for each instruction. Programmers wrote in assembly language and submitted their code to a tool called an *assembler* to translate the assembly language into machine code. Machine code is also called machine language.

Then, higher level languages were created. Nowadays, you can create complex control flows without even specifying their details; you can just indicate your desired results. These programs require a compiler to turn source code into machine code. The compilers can perform quite intelligent transformations and optimizations.

Many languages have multiple compilers available. You can find a choice of compilers for very popular languages, such as C and Java. In the free and open source community, most people use either the GNU Compiler Collection (GCC) or the LLVM compiler for C and C++.

Originally, a compiler would compile each file of source code to produce an intermediate *object file*, then combine all the object files into one program by invoking another tool called a *linker*. Modern compilers can compile multiple files at once in order to perform optimizations that cross the boundaries of files.

Compilers used to compile into assembly language, which could be useful because each type of computer processor supported a different machine code but may support the same assembly language. There are multiple variants of assembly languages. The last step in compilation and linking was to produce machine code. As with linking, modern compilers know how to assemble the code into machine code.

But there is another stage in many modern languages: intermediate code, usually called *byte code*. This code has been compiled into a binary format that is high-level enough to be portable. For instance, the Java language is compiled into byte code so that the program can be loaded onto many different types of processors.

In producing byte code from source code, the compiler has done much of the work. Each computer then hosts its own version of a program called a *virtual machine* that completes the transformation from byte code into a set of instructions that the virtual machine can execute. Sometimes, byte code is compiled into machine code, too. Going from byte code to a set of instructions is more convenient for end users than going from a high-level programming language to machine code. This was a great advantage for Java when it was invented, because its designers wanted it to run within a virtual machine plug-in for web browsers, where the user's processor and operating environment could be quite varied.

Java designers promoted the benefits of byte code through the marketing phrase “Compile once, run anywhere.” Some other advantages of byte code emerged later. New languages could be created that provided a very different experience to programmers (hopefully making the job of programming easier and producing more maintainable code) while creating the same byte code that Java virtual machines supported. Functions from these languages were easy to mix into existing Java programs.

Finally, there are some languages, such as Python, for which you don’t have to compile source code at all: You simply enter statements into a processing tool called an *interpreter*, which converts the code directly into machine code and executes it. Interpreters are slower than compilers, but some have become efficient enough that they don’t impose much of a performance penalty. Still, some popular libraries in the Python language include functions that are accessible to developers in the interpreted language but are programmed in the C language, to speed up execution.

Many interpreted languages provide compilers too. They produce either byte code or machine code, which then executes more quickly than the interpreter.

There are build tools to help programmers manage files and functions. A complex program might contain hundreds of files, and the programmer will need to compile different combinations of files using different compiler options at different times (for instance, to support debugging). A build tool lets the programmers store different options and combinations of files, and easily choose the desired type of build. Maven and Gradle are common build tools for Java and related languages.

Code Generation Tools

The programmer doesn’t have to start coding with a blank screen. Recently, services have sprung up that generate code based on your plain-text description of what you want. This a form of generative AI, and like other such services, some people complain that it draws on the work of former programmers without compensation and produces less robust source code (so far). Aside from ethical controversies, many programmers say that automatic code generation has greatly improved their productivity.

One form of code generation that has been available in many programming languages for many years is *refactoring*. It examines a large program, which can easily evolve over time into a tangle of files and functions. Refactoring moves functions around to create a more logical structure to the program in the pursuit of improved maintainability and reduced duplication of source code.

Some programmers are tasked with reproducing the functioning of other code. They may need to write a new program to replace a legacy program with missing source code that must be retired. Or they might be mimicking a competitor’s program. This kind of research is called *reverse engineering*. One useful tool for this purpose is a *disassembler*, which turns machine code into assembly language. There are also disassemblers for byte code.

Debuggers

Most programmers spend more time debugging than coding. People just don't think perfectly logically, and therefore forget some detail that the computer requires to run the program the way you want. Thus, your code is likely to fail at first try, and you can benefit greatly from a *debugger* to uncover the error.

A debugger supports intensive research efforts that can cut hours off of the process for finding errors.

A programmer can ask the program to stop at some key place in the program (a *breakpoint*), such as the beginning of a function or loop. The debugger can display the values of variables and even computer registers at the current point in the program. The programmer can also run each statement, one at a time, and see the results (*single stepping*). If the programmer wants to see when and how a variable changes during the run, they can set a *watchpoint*.

The most prominent debugger in the free and open source world, particularly for C and C++, is the GNU debugger, which works with the GNU compiler mentioned earlier. Other languages also have dedicated debuggers.

Analytical Tools

Although debugging can usually uncover bugs fairly quickly, it's better to eliminate spelling errors and other basic problems earlier in the programming process. Many ingenious analytical tools exist to check a program. *Static analysis* examines the code of a program. *Dynamic analysis* runs a program and checks for problems during its execution.

One of the earliest forms of static analysis was called a *linter*. It can detect, for instance, if you assign the value of a floating-point variable to an integer. This might or might not produce problems, and might or might not be caught by the compiler. In production, it could lead to incorrect results.

Nowadays, most compilers can do the job of a linter. Some compilers, such as the one for the Rust language, are notable for their strictness in rejecting poorly written code.

Many other types of analytical tools run separately from the compiler. For example, security analysis tools can find problems that make a program vulnerable to hacking. A common error by developers, for instance, is when their code calls a function and doesn't check whether that function returned an error.

Integrated Development Environments (IDEs)

Many programmers use text editors to enter and edit their code. Text editors are different from word processors, which introduce a lot of formatting (such as italic and bold, bullets and numbered lists, etc.). The text processor produces unadorned text, which programming languages require.

There are also sophisticated tools dedicated to helping programmers develop their programs. These tools are alert to the programming language in use. For instance, if you start to type the name of a variable of function, the tool can suggest a completion. These tools can check for errors while you're coding, format the code in a consistent and pleasing way, run analytical tools and debuggers, compile the code, handle check-ins to version control systems. and take care of other tasks for you. Therefore, they are called *integrated development environments* (IDE).

Eclipse is a popular open source IDE.

Common Types of Software Testing

Testing is a major part of software development. Programmers run unit tests as they develop the code. Other types of testing typically run when a programmer checks code back into the core repository, or during deployment.

Unit Testing

We've seen that a programmer should take great care to find errors before submitting code for integration into the core repository. Unit tests are crucial to catching errors.

Writing these tests is both an art and a science, and the volume of test code might exceed the volume of production code. There is even a development model called *test-driven development* (TDD), where programmers write tests before writing the code they want to test. Proponents of TDD claim it plugs gaps in testing and helps ensure that the code does what the programmer wants it to do.

It's important to test for things that go wrong during program execution, as well as things that go right. If the user, or another part of the program, submits invalid input to a function, it's important for the function to catch the problem and report an appropriate error message.

JUnit is a popular open source tool for running unit tests on Java programs.

Integration, Regression, and Smoke Testing

While unit tests focus on the individual actions of particular functions, the team should also run tests at a higher level to make sure the product, as a whole, works properly. The tests generally imitate user behavior. For instance, in a restaurant management application, tests could check whether the user gets the item they requested.

When a change to a program breaks some function that worked before, the failure is called a *regression*, and the tests are called *regression tests*.

As a team prepares a product for release, the first stage of testing is often very short. The team checks for the most important activities performed by a program and stops testing if any errors arise, thus saving time. This kind of testing is called a *smoke test*, because an application that malfunctions so easily is like a bad device that catches on fire.

Some products involve user interaction; web and mobile applications are common examples. Therefore, tools have been created to emulate user interactions. The test runs automatically, triggering the code that would have run if a user pressed the button. Selenium is a popular tool in this category.

Acceptance Testing

Programs with a user interface need an extra level of testing beyond proving that they react properly to certain inputs. The programs must also look right on the screen. *Acceptance testing* checks the impact of the program on the user. Quality assurance teams generally run these tests after integration and regression testing demonstrate that the program is formally correct and meets user expectations.

Security Testing

Security is obviously critical, and even a tiny security failure can expose an organization to major damage. We've seen that programmers can run analytical tools to check the security of the code. In the quality assurance stage, tests can also determine whether the program has vulnerabilities. The tests run the program with malicious inputs and make sure the program rejects the input without failing, performing unintended actions, or revealing sensitive information.

One kind of test that has proven valuable in some situations is *fuzz testing*. The test framework simply generates strings of random garbage and submits them as input to the program. This might seem to be a waste of time, but often turns up vulnerabilities that ordinary testing does not.

Performance Testing

After the program is judged by other tests to work correctly, teams should determine whether it runs fast enough. *Performance testing* requires an environment similar to the ones where the users will interact with the program. For instance, if users will submit requests from a long distance over a network, performance testing should also be done over a long-distance network.

Some programming libraries are tested through *benchmarks*: standard tests used to compare different libraries or different versions of the same library.

Common Deployment Environments

A CI/CD tool allows sophisticated ways to construct pipelines. Like computer programs, a pipeline can contain tests and branches. By branching, you can perform one set of activities in the test environment and another in the production environment. You can use the tool to automatically install the correct database or other software needed for different programs.

CD overlaps with DevOps. In cloud environments, CD and DevOps tools create the virtual computer systems in an automated fashion with all the components needed for the programs to run. Automated tools (sometimes called *orchestration* tools) check for the failures of virtual systems and automatically restart them.

The main job of the CD tool is to launch and step through each pipeline. The tool checks the results of each stage of the pipeline, and chooses whether to proceed or stop. The tool also allows scheduling, and logs its activities.

Often you have to run a task repeatedly with minor variations. For instance, teams distinguish between deploying to a test environment and deploying to production. Therefore, CD tools provide parameters that you can fill in with different values when you run the pipeline.

Sometimes, a process requires commands that traditionally were entered at the terminal. Thus, a CD tool provides mechanisms to run arbitrary commands. Usually, it provides hooks such as `preprocess` to run commands before a stage of the pipeline and `postprocess` to run commands after a stage of the pipeline.

Jenkins is probably the most popular open source tool for the orchestration described in this section.

Guided Exercises

1. What are some ways to check a program's security?

2. Why would you write multiple unit tests for a single program function?

Explorational Exercises

1. Your team has inherited an old application that runs slowly and is hard to add features to. What are some ways to improve the application, without throwing it out and writing a new one from scratch?

2. In large projects, frequently, Team A wants a feature implemented in a part of the system maintained by Team B, but Team B doesn't see the feature as a priority. How can Team A code the feature as part of Team B's project?

Summary

This lesson has laid out the types of tools used throughout development: compilers and other code generation tools, analyzers, tests, and CI/CD tools that automate integration and delivery. There are many options for each of these activities, and a tool that is popular today may be replaced in a year. Understanding how these tools all fit together in the development process helps you identify what you need.

Answers to Guided Exercises

1. What are some ways to check a program's security?

   First, human experts can examine the code.

   Many static and dynamic analysis tools exist to catch poor programming practices that expose a program to security threats.

   Other tools submit malicious input to running programs and check their reactions.

2. Why would you write multiple unit tests for a single program function?

   A program function usually has to run on many varieties of input, and each variety deserves its own test. For instance, the function might handle an input value of zero in a special way. You also need to anticipate invalid input and write tests to show that the function handles it appropriately.

Answers to Explorational Exercises

1. Your team has inherited an old application that runs slowly and is hard to add features to. What are some ways to improve the application, without throwing it out and writing a new one from scratch?

   First, make sure the project is under version control, if it wasn't placed there already.

   After adding a new feature, run regression tests to determine where the program fails, and assign programmers to figure out what functions are responsible. These functions can be selectively replaced.

   Performance testing can identify particular functions that run slowly, so you can focus your efforts on fixing or replacing the most inefficient code.

   If the code is in a language that is no longer popular, consider adding features in a language preferred by the team. Make sure that functions in the new language can be integrated with the old functions.

2. In large projects, frequently, Team A wants a feature implemented in a part of the system maintained by Team B, but Team B doesn't see the feature as a priority. How can Team A code the feature as part of Team B's project?

   Team B can allow Team A to create a new branch, code the feature, and submit the branch to Team B to merge into its project. Team A must not be empowered to do anything they want, however. Team B should provide documentation and help for Team A to follow project standards. A member of Team B must also review Team A's submission, and run all the usual forms of integration testing.

   This form of collaboration is sometimes called InnerSource, because it resembles open source but takes place within a single organization.

056.2 Source Code Management

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 056.2

Weight

3

Key knowledge areas

- Understanding source code repositories (public and private)
- Understanding the principles of source code management and repository organization
- Awareness of common SCM systems (Git, Subversion, CVS)
- Awareness of the terms Version Control System (VCS), Revision Control System and Source Code Management systems (SCM)

Partial list of the used files, terms and utilities

- Source code repositories
- Commits, Branches and Tags
- Feature, development and release branches
- Subrepositories
- Code merges

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	056 Collaboration and Communication
Objective:	056.2 Source Code Management
Lesson:	1 of 1

Introduction

Anyone who has ever edited a text document in a team knows the problems with such collaboration: Which version is the current one? Where is this version saved? Is it currently being edited by someone? Who has made which comments or changes to the text — when and why? The result is often differing versions of the document and, in the worst case, a collection of versions that nobody has a grasp of.

Now imagine a software project with hundreds of files, on which developers from all over the world are working by developing new features, fixing bugs, splitting off parts and developing them separately, and so on. Such a development process is no longer manageable without suitable tools.

Special software for *source code management* (SCM), also known as *version control systems* (VCS) or *revision control systems* (RCS), provides a remedy here. It eliminates the problems that were just outlined.

In the world of software development, SCM stands as a fundamental pillar, safeguarding the integrity of your codebase. Picture it as a diligent guardian, meticulously tracking every tweak and turn made to your source code over time.

Source Code Management System and Repository

The source code management system is the heart of a software project. Although important work goes on in discussion forums and other places, the SCM system represents both the history of the project and its current state as a living entity.

The source code *repository* is a kind of digital workshop for a project. Just as a physical workshop stores all the tools and items needed to manufacture a workpiece, a source code repository stores all the files, documents, and code related to a software project. It provides a structured environment for organizing and managing the project's assets.

Information is normally stored in the form of a directory tree. SCM systems typically use a client-server model, where any user can *pull* data from the repository as well as *push* data into it. The system keeps track of who made each change and when it was made, ensuring transparency and accountability within the team.

Imagine you're working on a project with multiple developers, and a bug is discovered in the code. With an SCM, you can easily examine the changes between versions to pinpoint the specific code change responsible for the bug.

Because the system remembers each version of the files as they change, a user has access to any of these versions and can revert to earlier versions (in case incorrect changes were made). So the repository is also a kind of archive, granting access to every change ever made to the project, and to the status of the project at any moment in its history.

SCM systems save space by keeping track of changes to a file instead of storing the complete file each time a change is made. This efficient storage method ensures that historical versions are accessible without consuming excessive resources.

Many tools, such as continuous integration/continuous delivery (CI/CD) and testing, revolve around the SCM system. People also build their reputations through the system by making their contributions visible to everyone. Therefore, source code management is not just about tracking changes; it's about preserving the integrity of your codebase and fostering collaboration among developers, ensuring that your projects can evolve in a dynamic and ever-changing landscape.

Popular SCM systems include Git, Subversion, and CVS. Like many other software functions, SCM is now often provided by specialized vendors. In other words, it is Software as a Service (SaaS) or a "cloud" service: Participants have the software on their local systems to manage their personal changes, but upload their changes to a central repository that benefits from typical cloud features such as 24/7 uptime, backups, and secure access.

Millions of developers now use cloud services, notably GitHub. GitLab is an alternative based on open source code. Both GitHub and GitLab allow people to work in the vendor’s cloud repositories or set up a local version of the SCM system. One advantage of working on those systems is the reputation one can gain through their “star” systems, which allow users to rate each other’s work. The cloud services add extra attractions such as change request trackers, rating systems, wikis, and discussion forums.

Repositories can contain both personal and corporate projects, meaning they are not necessarily exclusive to enterprise use. Any developer who wants to start a development project or work on an open source project, copying and modifying it according to their needs, can do so. In all these cases, it is important to have firm control over who can access your repository.

Understanding the terminology surrounding version control and source code management is essential in navigating its use. In the following sections, we take a closer look at some of these concepts and terms.

Commits, Tags, and Branches

A developer creates a *commit* each time a change is uploaded into the respository. Commits represent snapshots of changes made to the codebase at a particular point in time. Each commit includes metadata such as the author’s name, a timestamp, and a descriptive message explaining the changes. Commits help developers track the evolution of the codebase and understand the history of specific changes.

Consider a team of developers working on a web application. Each time they make changes to the codebase, they create a commit to document those changes. For instance, someone who adds a new feature to the application might create a commit with a message like “Added user authentication feature.” This commit captures the state of the codebase after the feature was implemented.

When a developer fixes a bug, the commit message usually refers to the number of the bug in the project’s bug tracking system.

Tags are named references to specific commits. They typically mark significant points in the project’s history, such as releases or milestones. Tags provide a way to label and refer to important versions of the codebase, making it easier to manage and navigate the project’s history.

Branches come into play when developers need to work on different tasks concurrently. Branches are independent lines of development that diverge from the main codebase. They allow developers to work on features or fixes in isolation without affecting the main code until they are ready for integration. Branches help organize development efforts and facilitate collaboration among team members.

For example, if one developer is working on adding a new feature to the application while another is fixing a bug, they can each create separate branches to isolate their changes. Once their work is complete, they can *merge* their branches back into the main codebase (Branches).

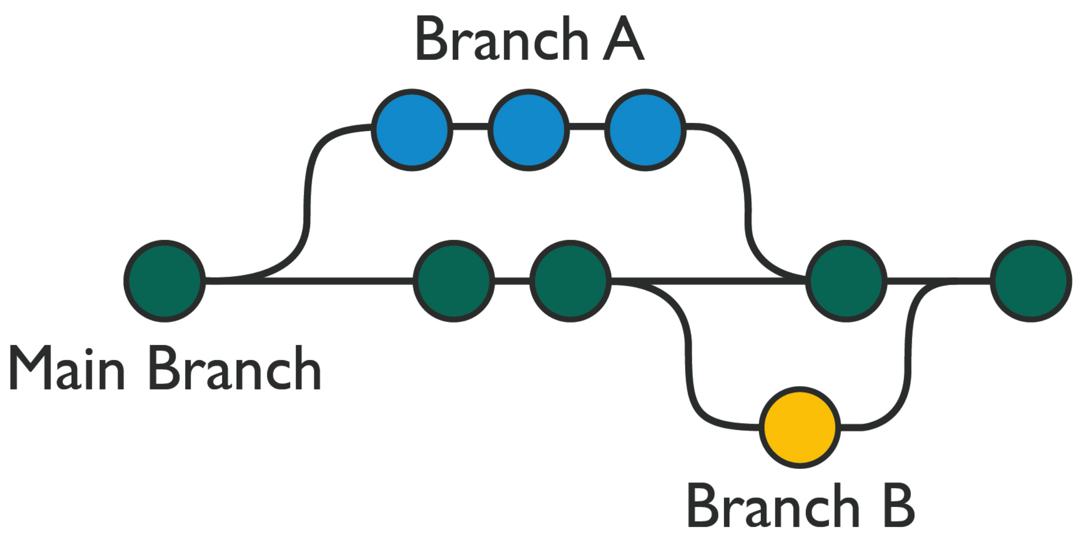


Figure 16. Branches

When multiple people are working on the same file, or are checking files into another branch, sometimes it will happen that two people made a change to the same line of a file. When the second person to make a change tries to check it in, the system warns of a *conflict*.

Some VCSes simply don't let a developer check in a file if it contains a conflict with the current version in the repository. The developer must check out the current version and figure out how to resolve the conflict, then check in a new version.

Cloud-based systems offer *merge* or *pull requests*. These are created by a developer who has worked on a local system or branch and believes their changes are suitable for inclusion in another branch. The developers for the target branch decide whether to accept the request.

Subrespositories

It often happens that the code of another, independent project (e.g., a media player) is required for the development of a software project (e.g., a complex website). Instead of copying the code of the media player in part or in full into your own project, many VCSs offer the feature of *subrepositories*, also known as *submodules*.

In our example, the repository of the media player is integrated into the repository of the website as a subrepository. It then appears as a separate directory in the directory tree. This means that the code base of the media player is fully available, but remains independent. If required, it can even be updated from the original repository of the media player.

This capability proves valuable for handling intricate projects with numerous dependencies or integrating third-party libraries and frameworks into a codebase. Submodules enhance project organization and facilitate collaboration by allowing developers to work with interconnected codebases efficiently.

General Use of a Source Control Management System

Each participant in a project starts by creating an identity, normally tied to the participant’s unique email address. A cloud-based system manages identities through accounts, as social media sites and other organizations do.

New developers go through a sequence like the following, when using an SCM system:

1. Install the SCM software, if it is not already provided on their operating system.
2. Get the entire project onto the local system one time, also known as *cloning*.
3. From this step onward, work locally (in one or more branches, if necessary), and push changes to the repository.
4. Pull the recent version from the repository before the next working session.

Project managers decide whom to trust and give access to the repository. Senior devoélopers have the important responsibiilty of deciding when changes submitted by other contributors are ready to go into the main branch of the respository.

On cloud-based systems, a project owner can control the accessibility of a repository by setting its visibility to either public or private. Public repositories grant read access to anyone on the internet. Private repositories, however, limit access solely to the owner, individuals with whom the owner have explicitly shared access, and, in the case of organization repositories, specific members of the organization.

Imagine a team of developers working on an e-commerce website. They decide to add a new feature that allows customers to track their orders. To implement this feature, they create a *feature branch* with a name such as `order-tracking`, where they can work on the necessary code changes without affecting the main codebase. Once the feature is complete and tested, they merge this branch into the main *development branch* for further integration and testing.

The main development branch serves as a central hub where all the new features are brought together for testing. For instance, if multiple developers are working on different features concurrently, they can merge their feature branches into the development branch to ensure that everything works together smoothly. This integration process helps identify and resolve any conflicts or compatibility issues early on.

When it’s time to release a new version of the e-commerce website, the team creates a *release branch*, such as `v2.0`, from the development branch. They focus on stabilizing the codebase, fixing any last-minute bugs, and conducting thorough testing to ensure a smooth release. Once the release is ready, the code from the release branch is deployed to production, and the cycle begins anew.

Common Version Control Systems

Some of the best-known version control systems are Git, Subversion (also known as SVN), and CVS. All of these are open source.

Git is a distributed version control system widely used in software development and other fields. When using Git, each developer has a complete copy of the codebase on their computer.

This decentralized approach enables developers to work offline and collaborate seamlessly, without relying on a central server. That is, developers can work independently on different features or fixes and merge their changes together seamlessly. Even if the central server goes offline, developers can continue working and share updates with each other.

Consider the development of the Linux kernel, which initially relied on a centralized version control system called BitKeeper. When BitKeeper’s free-of-charge status was revoked, Linus Torvalds and the Linux community developed Git as a distributed alternative. This decision enabled non-linear development and the efficient handling of large projects such as the Linux kernel. The success of Git for the Linux kernel — an extremely complex project with thousands of developers and innumerable branches — shows the power and scability of Git.

Most software development these days uses Git. Extremely popular SaaS offerings are built around it.

Git treats conflicts by giving the developer a file with both versions of the changed line, clearly marking which version of the file the lines are from. The developer must decide how to resolve the conflict and check in a coherent version of the file.

Subversion (SVN) was probably the most popular SCM system before Git was invented. Unlike Git, Subversion is centralized: The version history resides on a central server. Developers connect to this server to make changes, ensuring that everyone is working with the latest version of the codebase.

Assume you are part of a team working on a project using Subversion. Each time you need to make changes to the codebase, you connect to the central SVN server to check out a working copy of the code. This ensures that you’re working with the most up-to-date version of the project. After making your changes, you commit them back to the server, updating the central repository with your modifications. The centralized workflow helps maintain consistency and ensures that everyone is working towards the same goals.

Before Subversion, *CVS* was a very popular, centralized version control system. It had design problems that led to the design of Subversion as an alternative.

Guided Exercises

1. Name three core features of SCM systems.

2. Describe the concept of tagging in SCM systems and explain why it is important for managing software releases.

3. What is the difference between a branch and a subrepository in a SCM system?

Explorational Exercises

1. Compare Git and Subversion (SVN) in terms of their architecture and workflow.

2. What is the “index” or “staging area” in Git?

3. Outline the Git trunk-based branching strategy.

4. Which of the following SCM systems are open source?

Git	
Mercurial	
Subversion	
GitHub	
Bitbucket	
GitLab	

Summary

This lesson explained the central role of source code management systems in modern software development. You learned the basic terms and ways of using the system, including repository, branches, tags, and merges.

Answers to Guided Exercises

1. Name three core features of SCM systems.
   - Logging changes to the source code
   - Management of (simultaneous) access to the source code by developers
   - Ability to restore any state of development of the files or the entire project
2. Describe the concept of tagging in SCM systems and explain why it is important for managing software releases.

   Tagging is the practice of assigning descriptive labels or names to specific commits within the codebase, serving as markers for significant points in the project's history, such as releases or milestones. These tags offer a convenient means of referring to particular versions of the codebase and monitoring the project's evolution over time.

3. What is the difference between a branch and a subrepository in a SCM system?

   A branch is a parallel development line in a project, such as for bug fixing or developing new features, which is usually merged back into the main development branch as soon as the task has been completed. A subrepository or sumbmodule is an independent project whose repository is integrated into a project in order to access its code base. The subrepository appears as a directory in the directory tree of the project and remains independent.

Answers to Explorational Exercises

1. Compare Git and Subversion (SVN) in terms of their architecture and workflow.

   Git is a distributed version control system (DVCS), allowing each developer to have a complete copy of the codebase and work even offline on the source code. Git has gained widespread popularity among developers due to its speed, flexibility, and robust branching and merging capabilities. SVN is centralized VCS, with the version history residing on a central server. It remains popular in certain enterprise environments due to its centralized nature and mature feature set.

2. What is the “index” or “staging area” in Git?

   The index or staging area is an intermediate layer between the local working copy of the project and the current version on the server. It is a file in which all the information for a user’s next commit is stored.

3. Outline the Git trunk-based branching strategy.

   Trunk-based development is a strategy that emphasizes frequent integration of changes into the main codebase (trunk). Developers work on short-lived feature branches and merge them into the trunk multiple times a day, ensuring continuous integration and rapid feedback.

4. Which of the following SCM systems are open source software?

Git	X
Mercurial	X
Subversion	X
GitHub	
Bitbucket	
GitLab	X

056.3 Communication and Collaboration Tools

Reference to LPI objectives

Open Source Essentials version 1.0, Exam 050, Objective 056.3

Weight

2

Key knowledge areas

- Understanding common tools for communication
- Understanding common ways to capture and secure knowledge
- Understanding common tools for information management and publication
- Understanding common types of documentation
- Understanding common collaboration features of source code management platforms
- Understanding the concepts of stand-alone, federated and centralized managed applications and platforms

Partial list of the used files, terms and utilities

- Instant messengers
- Chat platforms
- Mailing lists
- Newsletters
- Issue trackers and bug trackers
- Bug reports
- Merge requests and pull requests
- Helpdesk and ticketing systems

- Wikis
- Document Management Systems (DMS)
- Documentation websites
- Product websites
- Content Management Systems (CMS)
- Architecture documentation
- User documentation
- Administrator documentation
- Developer documentation

Lesson 1

Certificate:	Open Source Essentials
Version:	1.0
Topic:	056 Collaboration and Communication
Objective:	056.3 Communication and Collaboration Tools
Lesson:	1 of 1

Introduction

Many open source projects have active participants worldwide. Collaboration mostly happens in “virtual space,” and the people involved are often distributed across countries, continents, and timezones. Furthermore, they usually speak different native languages. This means you can be located anywhere in the world to make a contribution to an open source project, no matter your country or language!

While this diversity makes contributing to an open source project extremely rewarding, as you can widen your scope and learn a lot, at the same time it can make communication and coordination quite challenging. Efficient and effective communication is key to the success and sustainability of any open source project. To ensure good communication, open source projects have created structures and use a variety of tools that help to make contributions easy and the cooperation more effective.

Another challenge is that open source projects, often driven mostly by volunteers, face some fluctuation in participation. People’s lives and hobbies change, and some might lose interest or just run out of time. You might contribute to an open source project for years, but when you change your job, get married, or start raising kids, you might not have enough time anymore to volunteer.

Therefore, in addition to providing efficient means of communication via proper tooling, open source projects need to ensure the preservation and sharing of knowledge to avoid “reinventing the wheel.” Information preservation also helps contributors learn from the past mistakes of other contributors and avoid making the same errors.

This lesson presents common tools for cooperating in an open source project and introduces you to ways of communicating in an international community. You’ll discover that there’s an easy way for everyone to make their first contribution!

As an example of communications and information sharing, we will look at the *LibreOffice*. LibreOffice is an open source office productivity suite that is available in more than one hundred languages. Its end users range from the casual home user to large governments and corporations. Likewise, there’s a wide range of contributors — not only developers, but also localizers, documentation authors, marketing people, quality assurance engineers, infrastructure administrators, graphics and UX designers, and many, many more.

In other words: For LibreOffice, as for many other open source projects, you can bring in your skill and talent in the area you are comfortable with. You don’t necessarily need to be a technically trained person or a developer — you can bring in your creativity and artistic talent or your language skills as well. The project has also launched a website that shows the various areas of contributions: https://whatcanidoforlibreoffice.org. So LibreOffice will nicely illustrate many aspects of community work.

Ways to Communicate

Before looking into the details of communication tools, it is important to understand how communication works in general, as these considerations inform the choices of tools.

Open source projects enjoy two different major ways to communicate. With *synchronous* communication, people communicate at the same time. Examples include of course direct conversations, but also video conferences or phone calls. With *asynchronous* communication, people communicate at different times. Examples are email and SMS, but also a postal letter or a telefax.

This distinction is not always easy to make, however. As one example, a messenger application on a mobile phone like WhatsApp, Telegram, Signal, or Element technically is an asynchronous way to communicate. If, however, both conversation partners are online at the very same time and reply instantly, they enter a direct conversation.

This example shows that a part of communication also depends on how people use their tools and what expectations they have. Each task might require a different set of communication tools. The following sections will explain these ideas in detail.

Synchronous Communication

Synchronous communication is a very effective but also very demanding way to communicate. It brings people together at the same time in the same physical or virtual space.

A direct meeting is ideal for talking things through interactively instead of sending long email messages that come with the risk of misunderstandings. Imagine you want to learn more about an open source project and get to know the people from the community. Email messages and websites can help make an introduction and have a lower barrier to entry, but a truly meaningful first impression is made when you can actually talk to someone in person — it's these direct interactions that usually fascinate people about an open source project and make them want to contribute.

Apart from getting to know each other, synchronous meetings are also great to talk things through interactively, e.g. in case of conflicts or problems, or when you have to share a negative message.

The downside is that international projects face a challenge that technology can't overcome: timezones. If you have contributors on different continents, it will be very hard to find suitable meeting slots that work for everyone. Someone in Australia might just be waking up when someone in Europe is close to calling it a day. As an additional challenge, some people can work only at night or weekends, while others prefer office hours during business days.

Additionally, English is not a language everyone speaks fluently, and there are no widely accessible live translation tools available yet, which brings an additional barrier.

Even so, video conferences are one of the most frequent tools of communication in open source projects. The LibreOffice project, which serves as an example for this lesson, has regular online meetings for its community, e.g. for developers, marketing, infrastructure, quality assurance, user experience, and design.

Asynchronous Communication

Asynchronous communication lets you participate in the discussion at a time and speed that is convenient for you. The most known example is probably an email message: You can reply to a message whenever you prefer, whether it's the next minute, the next day, or the next hour.

For asynchronous communication, the content usually is in writing, which additionally opens up the possibility of machine translation. This helps you to read and understand messages written in a language different from your native language, and you can even translate back your reply.

Additionally, content that is already written down makes it much easier to retain knowledge and produce documentation. Imagine you want to write a support document about how to use a specific feature of a software. If you explain it to a user over the phone, it will be much harder to turn your speech into a proper documentation page than when you explain the procedure in written text.

Internal Versus External Communication

Last but not least, communication also depends on the intended recipients, i.e. whether it is internal or external. You will write an internal, technical note to the system administrators differently from how you write a press release that gets sent to hundreds of journalists. Keep in mind, however, that due to the nature of an open source project, communication that might not be explicitly intended for a public audience will nevertheless be publicly visible, for example in mailing list archives (in the case of LibreOffice, it's https://listarchives.documentfoundation.org/).

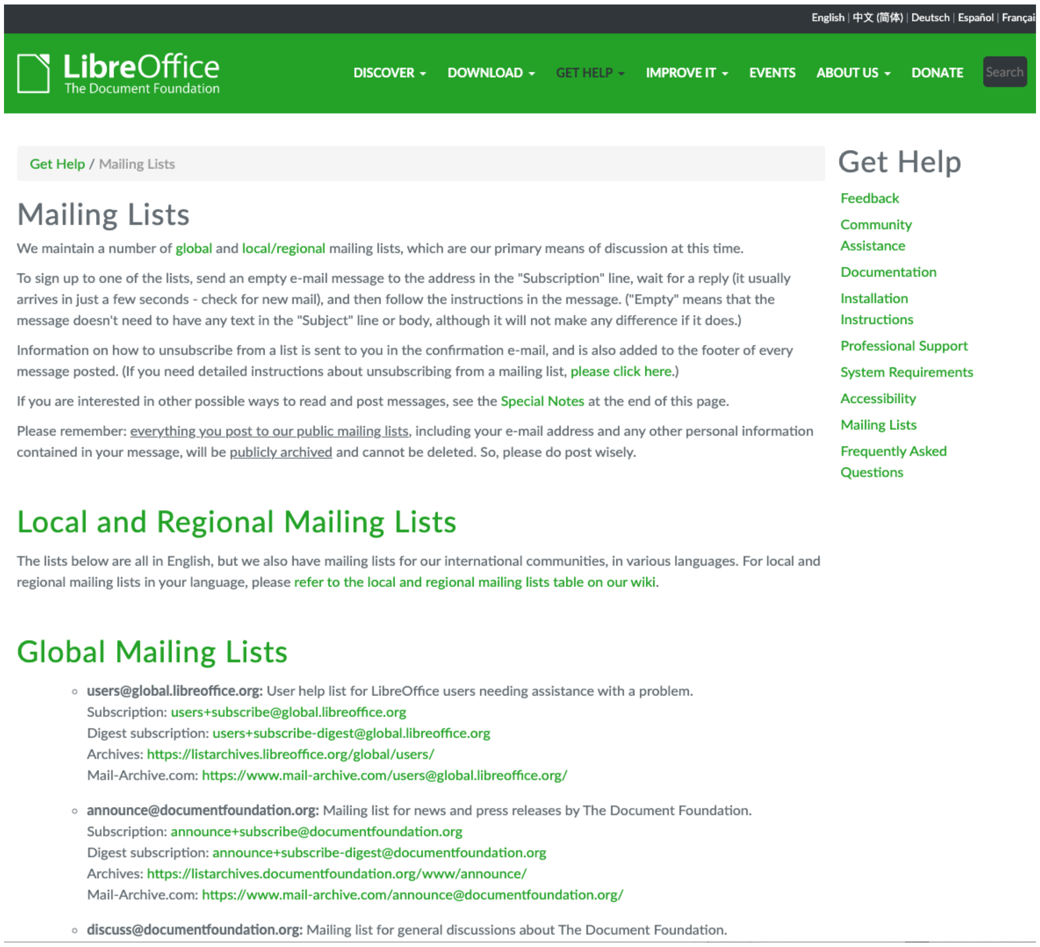
Tools for Communication

With these different general aspects of communication in mind, you will learn in the following sections about several tools for communication that are commonly used in an open source project.

Email, Mailing Lists, and Newsletters

One of the first tools you will encounter when you participate in an open source project is the "classic" email. Many projects run *mailing lists*, which are essentially distribution lists for email: With one email message, you can reach hundreds or even thousands of subscribers who are interested in particular topics.

Mailing lists are among the oldest known tools in any open source project and are used for coordination internal to the project as well for interacting with users. If you have a question about the software, or want to report an error in the program, chances are high that there is a mailing list that allows you to do that. The LibreOffice community, for example, offers a variety of international and local mailing lists for various topics (https://www.libreoffice.org/get-help/mailing-lists/), ranging from user support to developer discussions and infrastructure coordination (LibreOffice mailing lists web page).



Mailing Lists

We maintain a number of global and local/regional mailing lists, which are our primary means of discussion at this time.

To sign up to one of the lists, send an empty e-mail message to the address in the "Subscription" line, wait for a reply (it usually arrives in just a few seconds - check for new mail), and then follow the instructions in the message. ("Empty" means that the message doesn't need to have any text in the "Subject" line or body, although it will not make any difference if it does.)

Information on how to unsubscribe from a list is sent to you in the confirmation e-mail, and is also added to the footer of every message posted. (If you need detailed instructions about unsubscribing from a mailing list, please click here.)

If you are interested in other possible ways to read and post messages, see the Special Notes at the end of this page.

Please remember: everything you post to our public mailing lists, including your e-mail address and any other personal information contained in your message, will be publicly archived and cannot be deleted. So, please do post wisely.

Local and Regional Mailing Lists

The lists below are all in English, but we also have mailing lists for our international communities, in various languages. For local and regional mailing lists in your language, please refer to the local and regional mailing lists table on our wiki.

Global Mailing Lists

- **users@global.libreoffice.org:** User help list for LibreOffice users needing assistance with a problem.
  Subscription: users+subscribe@global.libreoffice.org
  Digest subscription: users+subscribe-digest@global.libreoffice.org
  Archives: https://listarchives.libreoffice.org/global/users/
  Mail-Archive.com: https://www.mail-archive.com/users@global.libreoffice.org/
- **announce@documentfoundation.org:** Mailing list for news and press releases by The Document Foundation.
  Subscription: announce+subscribe@documentfoundation.org
  Digest subscription: announce+subscribe-digest@documentfoundation.org
  Archives: https://listarchives.documentfoundation.org/www/announce/
  Mail-Archive.com: https://www.mail-archive.com/announce@documentfoundation.org/
- **discuss@documentfoundation.org:** Mailing list for general discussions about The Document Foundation.

Figure 17. LibreOffice mailing lists web page

Every message sent is usually also stored in a public *mailing list archive* for future reference. Once sent out, a message cannot be easily deleted. A common phrase goes, “The internet never forgets.” Therefore it is advisable to be careful what you write, because you probably can’t take it back. For example, some people have their private address or phone number in the signature, or send confidential documents as attachments, which you should carefully avoid.

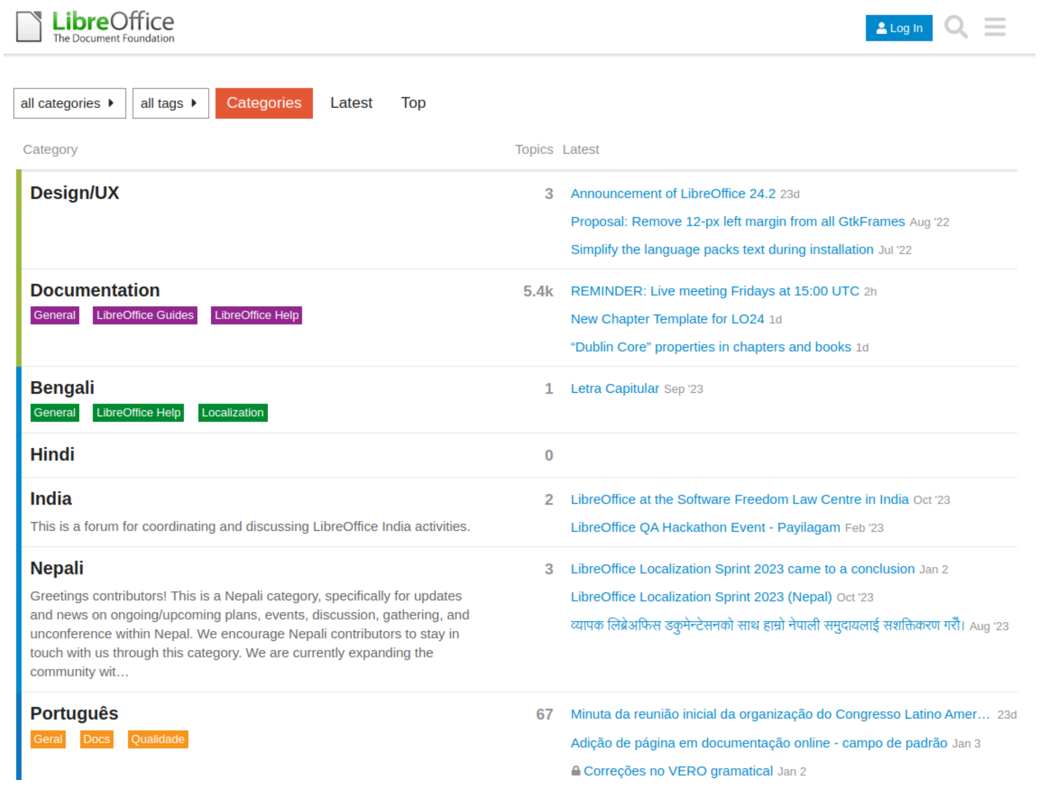
One disadvantage of mailing lists is that managing them in your mail program is not always straightforward. You will want to create so-called *filters* based on specific elements of the message, e.g. a prefix in the subject line. The subtleties of managing large quantities of email can be a barrier for unexperienced users — especially if your message is a one-off. Therefore, more and more projects migrate to discussions forums, about which you will soon learn.

A special form of mailing list is a *newsletter*. If you want to stay up to date about the latest project developments and get informed about new software releases, you can subscribe to the newsletter and get an email message when something important happens.

Discussion Forums

Apart from the overhead of managing mailing lists in your email client, another downside of email is that more and more people, especially the younger generation, is not so much into email communication anymore. For this and other reasons, more and more open source projects move their communication to *discussion forums*. The general idea is very comparable to email: Each forum has various categories with specific topics to engage in discussion, so-called *threads*. Similar to email, on a forum you can get in touch with the open source project and coordinate activities, make suggestions for the direction of the project, and report errors as an end user.

Everything posted to a forum is usually visible to the general public, just as in a mailing list; but unlike there, depending on the forum's configuration, contents can also be edited or deleted later. The usability of forums is, especially for inexperienced users, often better than mailing lists. The LibreOffice project began converting several of its mailing lists to forums (https://community.documentfoundation.org/) and has since seen an increase in participation in the discussion.



Category	Topics	Latest
Design/UX	3	Announcement of LibreOffice 24.2 23d Proposal: Remove 12-px left margin from all GtkFrames Aug '22 Simplify the language packs text during installation Jul '22
Documentation General · LibreOffice Guides · LibreOffice Help	5.4k	REMINDER: Live meeting Fridays at 15:00 UTC 2h New Chapter Template for LO24 1d "Dublin Core" properties in chapters and books 1d
Bengali General · LibreOffice Help · Localization	1	Letra Capitular Sep '23
Hindi	0	
India This is a forum for coordinating and discussing LibreOffice India activities.	2	LibreOffice at the Software Freedom Law Centre in India Oct '23 LibreOffice QA Hackathon Event - Payilagam Feb '23
Nepali Greetings contributors! This is a Nepali category, specifically for updates and news on ongoing/upcoming plans, events, discussion, gathering, and unconference within Nepal. We encourage Nepali contributors to stay in touch with us through this category. We are currently expanding the community wit…	3	LibreOffice Localization Sprint 2023 came to a conclusion Jan 2 LibreOffice Localization Sprint 2023 (Nepal) Oct '23 व्यापक लिब्रेअफिस डकुमेन्टेसनको साथ हाम्रो नेपाली समुदायलाई सशक्तिकरण गरौं। Aug '23
Português Geral · Docs · Qualidade	67	Minuta da reunião inicial da organização do Congresso Latino Amer… 23d Adição de página em documentação online - campo de padrão Jan 3 Correções no VERO gramatical Jan 2

Figure 18. LibreOffice forums web page

Instant Messages and Chat Platforms

Another way to get in touch with an open source community is through instant messages and chat platforms. With the rise of popular tools like WhatsApp, Telegram, Signal, and Matrix, nearly everyone has already installed one of these popular applications on their devices, which makes the entry barrier much lower. Instant messages are also much more popular with the younger generation than email or forums. It therefore comes as no surprise that many open source projects these days adopt them.

Participants in chats type messages that are sent to all other participants, similar to email. Depending on the chat platform, a message can be enriched with formatting, graphics, and attachments.

Structurally, message apps are organized similar to a forum or email. Several *groups* or *channels* are available, so you can join discussions on topics you are interested in. Messages can usually be edited or deleted, and often there are also announcement-only channels that have a function similar to email newsletters.

One drawback of instant messenger apps is that people usually install them on their phones, so they receive notifications of every message sent. This quickly can turn into information overflow or “alert fatigue.” With proper configuration, however, these notifications can be kept under control.

Another downside is that many messenger apps are proprietary and in the hands of one vendor. This makes preserving knowledge for the long term more complicated, if data is not freely accessible.

Stand-Alone, Federated, and Centralized Communications

Open source projects work in the open, based on open standards and open tools. Therefore it is critical to understand how the various tools are designed with regard to interoperability. The options can be split into three main categories.

A *stand-alone* platform runs in an isolated manner for one community. Examples are forums or wikis, which are usually not connected with instances of other projects.

Decentralized or *federated* systems run individually for each community, but can connect to each other. One example is email, because a local email server can send email to any other email server in the world. Other examples include Nextcloud and ownCloud, which can “federate” file shares with other servers, and the Element messenger service, with which you can communicate with users of other servers. The same principles apply to the Mastodon social network.

Both stand-alone and distributed platforms have one huge advantage: The open source project stays in full control over all content and functionality. All the knowledge stored in such a system remains the property of the open source community and is not subject to third parties.

A *centralized* system, on the other hand, is run by one provider and does not interact with third parties. Classical examples are social networks like Facebook or Instagram, or messenger apps like WhatsApp or Telegram. All content is stored on the external provider's servers and is subject to their terms and conditions.

If you are active in an open source community, you likely get in touch using all three of these options. Centralized systems are great for reaching out to people, as they often are popular and have a large user base. For actual work in the project, however, a federated or stand-alone system under the control of the community is best.

Tools for Collaboration

The distinction between tools for communication and tools for collaboration is not always straightforward. For the purpose of this lesson, the main goal of communication tools is to enable general communication between diverse participants, whereas the main goal of collaboration tools is to help people work together.

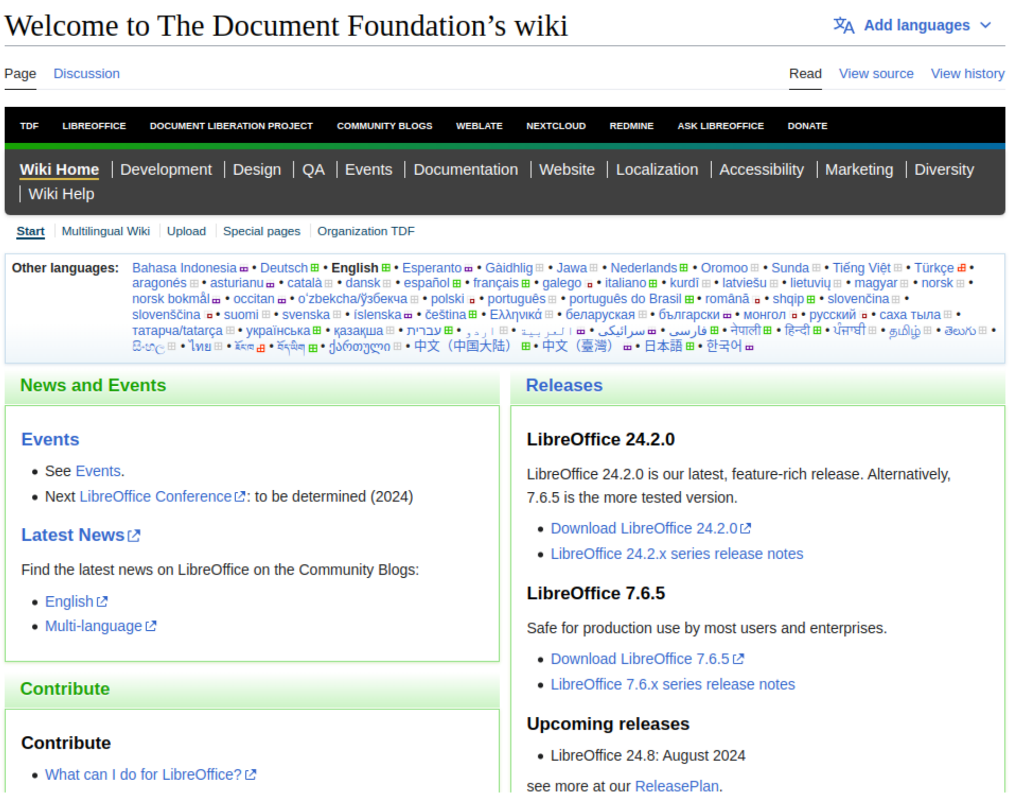
Proper collaboration tools allow file storage, real-time collaboration on documents, tracking of document versions and changes between software releases, and much more. While email or forums can serve as general-purpose storage, specialized tools make knowledge more easily accessible and collaboration much more effective.

In other words, these are specialized tools for specific tasks. If you want to contribute to an open source project, you will encounter them very soon.

Wikis

One of the oldest and most popular tools for collaboration is a *wiki*. Made famous especially by Wikipedia, a wiki allows users to work together on a website that is compiled of several documents or "articles." They can be grouped in various categories, filtered by language, and contain formatting, spreadsheets, and images.

Wikis are often used as a knowledge base where everybody can contribute. If you want to contribute content to an open source project, participating in their wiki is one of the most straightforward ways. You can take existing content and translate it into your native language, edit and update existing articles, or create new content. In the wiki of the LibreOffice project (https://wiki.documentfoundation.org), you can find marketing material, board meeting minutes, installation instructions, and conference planning, all in a variety of languages (LibreOffice Wiki).


Welcome to The Document Foundation’s wiki

Add languages

Page | Discussion | Read | View source | View history

TDF | LIBREOFFICE | DOCUMENT LIBERATION PROJECT | COMMUNITY BLOGS | WEBLATE | NEXTCLOUD | REDMINE | ASK LIBREOFFICE | DONATE

Wiki Home | Development | Design | QA | Events | Documentation | Website | Localization | Accessibility | Marketing | Diversity | Wiki Help

Start | Multilingual Wiki | Upload | Special pages | Organization TDF

Other languages: Bahasa Indonesia • Deutsch • **English** • Esperanto • Gàidhlig • Jawa • Nederlands • Oromoo • Sunda • Tiếng Việt • Türkçe • aragonés • asturianu • català • dansk • español • français • galego • italiano • kurdî • latviešu • lietuvių • magyar • norsk • norsk bokmål • occitan • oʻzbekcha/ўзбекча • polski • português • português do Brasil • română • shqip • slovenčina • slovenščina • suomi • svenska • íslenska • čeština • Ελληνικά • беларуская • български • монгол • русский • саха тыла • татарча/tatarça • українська • қазақша • עברית • اردو • العربية • سرائیکی • فارسی • नेपाली • हिन्दी • ਪੰਜਾਬੀ • தமிழ் • తెలుగు • සිංහල • ไทย • རྫོང་ཁ • བོད་ཡིག • ქართული • 中文（中国大陆） • 中文（臺灣） • 日本語 • 한국어

News and Events

Events

- See Events.
- Next LibreOffice Conference: to be determined (2024)

Latest News

Find the latest news on LibreOffice on the Community Blogs:

- English
- Multi-language

Contribute

Contribute

- What can I do for LibreOffice?

Releases

LibreOffice 24.2.0

LibreOffice 24.2.0 is our latest, feature-rich release. Alternatively, 7.6.5 is the more tested version.

- Download LibreOffice 24.2.0
- LibreOffice 24.2.x series release notes

LibreOffice 7.6.5

Safe for production use by most users and enterprises.

- Download LibreOffice 7.6.5
- LibreOffice 7.6.x series release notes

Upcoming releases

- LibreOffice 24.8: August 2024

see more at our ReleasePlan.

Figure 19. LibreOffice Wiki

Many open source projects also host their documentation and their program’s integrated help system in a wiki. Additionally, old versions of a page — called *revisions* — are archived for future reference.

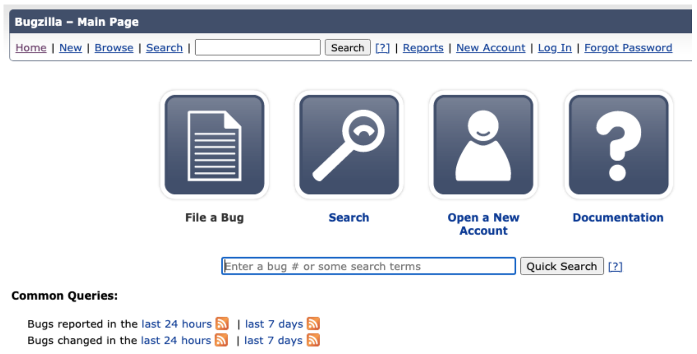
Although wikis can also store project files, there are better tools for this purpose, as you will learn in this lesson.

Bug and Issue Trackers

Another frequently used tool in an open source project are *bug trackers*, also called *issue trackers*. If you discover a problem in the software, or want to suggest a new feature, you might think of just sending an email message about it. With a dedicated tool like a bug tracker, however, you can provide all required information and steps to reproduce a problem in a structured way, which makes it easier for developers to reproduce the issue. Such a structured report is called a *bug report*.

In addition, the information is not lost and the project doesn’t forget about the issue; it can assign it to the right person and see how many bugs are being processed or fixed.

The LibreOffice project provides a bug tracker that is open for everyone to contribute (https://bugs.documentfoundation.org).


Bugzilla – Main Page
Home | New | Browse | Search | Search [?] | Reports | New Account | Log In | Forgot Password
File a Bug
Search
Open a New Account
Documentation
Enter a bug # or some search terms
Quick Search [?]

Common Queries:

Bugs reported in the last 24 hours | last 7 days
Bugs changed in the last 24 hours | last 7 days

Figure 20. LibreOffice bug tracker

Helpdesks and Ticketing Systems

A very similar tool is a *helpdesk* or *ticketing system*. Its focus is less on reporting software problems, but rather to help users with all kind of issues and requests, such as for the project's website.

The classic helpdesk is a customer service hotline. A customer calls and reports a problem, which is turned into a *ticket*. The workflow of a helpdesk system usually revolves around priorities, escalation levels, and response time.

Not all open source communities provide such a system, but many commercial companies do. For the LibreOffice project, for instance, there is a ticketing system for infrastructure to report problems with servers and web services.

Content Management System (CMS)

Another important tool for collaboration is a *content management system* (CMS). As the name suggests, it helps to manage the content, most notably for websites. If you want to contribute to the content or the design of a project's website, making yourself familiar with their CMS is the way to go.

CMS systems, similar to wikis, help to structure content in categories and languages. They often provide a WYSIWYG ("what you see is what you get") editor and embed everything in a proper template: Titles, headings, page layout, and menu entries are done automatically, so you can focus fully on the content. Additionally, in case of a page redesign, the content will not disappear, but will be adapted to the new template.

Document Management System (DMS)

A document management system is not to be confused with a content management system. While a CMS is designed to show content in a predefined template, like for presenting a website, a DMS is used for the management of documents such as contracts, invoices, receipts, email messages, and all other sorts of correspondence.

Another difference is that a CMS is often used for public presentation, while a DMS is mostly used to manage internal documents that are not intended for the general public.

As a contributor to an open source project, you are less likely to encounter the document management system, as that is often reserved for specific roles such aas accounting or legal.

Source Code Management (SCM)

If you are a developer in an open source project, one of the key tools you will soon run into is the source code management platform. Comparable to a wiki for documentation authors, SCMs are used by software developers to work together on the code.

Historical SCMs include CVS and Subversion, but nowadays Git is mostly used including by the LibreOffice community (https://git.libreoffice.org/). These tools are available on the command line, but there are also graphical interfaces available to make interaction easier, especially for beginners.

Source code management platforms track different versions of each file, handle changes and edits to the code, record who made which change, and ideally give a full history of the software’s development.

Developers can “check out” a specific state of the software, locally work on it, e.g. by fixing a bug or implementing a new feature, and then request that this change be added to the main development line of the software by means of a so-called *merge request*, also known as a *pull request*. Accepting the merge or pull request “merges” the changes done by one author with the main code.

There are centralized platforms (GitHub and GitLab) that integrate SCM with wikis, issue tracking, and other collaborative tools.

Source code management platforms are not strictly limited to program code. As one example, this very lesson was worked on collaboratively in a Git repository!

Documentation

Key to the success of every open source project is proper documentation, ideally in several languages. The LibreOffice project provides up-to-date books, guidelines, and reference cards (https://documentation.libreoffice.org) for its software, as well as individual help pages on specific functions (https://help.libreoffice.org).

In general, there are different kinds of documentation, which we'll discuss in the following sections.

User Documentation

Best known is the documentation for end users, which explains how to use the software. If you don't know about a feature or function in the program, the documentation — which can range from individual help pages to a complete book — is your first point of reference.

The documentation website, where the use of the software is explained, is often one of the most frequented websites next to the product website, which presents an overview of the software and its community.

Administrator Documentation

For use in larger environments, like deployments in a company, administrator documentation provides all relevant information for deploying the software on a larger scale. This includes connecting to user databases and file storage, centralized configuration management, and handling updates.

Developer and Architecture Documentation

Another category of documentation targets developers and is called developer documentation or architecture documentation. If you are a developer and want to contribute to the code of a program, this documentation tells you about the software architecture, the coding standards, and the tools and workflows used to work on the software.

The LibreOffice project has published a developers guide in its wiki to help interested developers join the community (https://wiki.documentfoundation.org/Documentation/DevGuide).

Guided Exercises

1. Why do open source projects in particular have to take care of proper tools for communication and collaboration?

2. Name one example each for synchronous and asynchronous communication.

3. What is a disadvantage of messenger apps and how can this be avoided?

4. Name two functions of a wiki.

5. What is the difference between a bug tracker and a helpdesk system?

6. What is the difference between a content management and a document management system?

7. What is the advantage of stand-alone or federated systems over centralized systems?

Explorational Exercises

1. What is one of the main differences between a local sports club and an international open source project?

2. Why can contributing to an open source project be particularly rewarding?

3. Which bug tracking software does the Ubuntu project use?

4. What is the name of the website for the Linux kernel mailing lists?

Summary

In this lesson, you've learned about a variety of tools used for communicating and collaborating in an open source project. You've heard the difference between synchronous and asynchronous communication and between decentralized, centralized, and stand-alone tools. You also learned why specific tools can be helpful for specific tasks to make contributing to an open source project fun and rewarding.

Answers to Guided Exercises

1. Why do open source projects in particular have to take care of proper tools for communication and collaboration?

   On one hand, working together in a worldwide distributed group has its set of challenges, which proper tools can help to address. On the other hand, volunteers might not stay indefinitely, so retaining and sharing knowledge is another important aspect in the use of communication and collaboration tools. Making contributions easy helps the sustainability of a project.

2. Name one example each for synchronous and asynchronous communication.

   Synchronous communication could be a direct conversation, a phone call, or a video conference. Examples of asynchronous communication include email, SMS, postal letters, and faxes.

3. What is a disadvantage of messenger apps and how can this be avoided?

   When installed on the phone, you might receive many notifications, one for each new message. Proper configuration can help avoid this. Another disadvantage is that many messenger apps are run by proprietary vendors.

4. Name two functions of a wiki.

   Collaborative editing and translation of articles.

5. What is the difference between a bug tracker and a helpdesk system?

   A bug tracker is a specialized software tool to report bugs or request features in software. A helpdesk system focuses on supporting inquiries and managing all kind of issues and requests, e.g. on the website.

6. What is the difference between a content management and a document management system?

   A CMS is used for presenting content in a specific way or template, mostly publicly. A DMS is used to store existing correspondence, mostly internally.

7. What is the advantage of stand-alone or federated systems over centralized systems?

   A centralized system is under the control of one external provider. All content is stored on the external provider’s servers and subject to their terms and conditions.

Answers to Explorational Exercises

1. What is one of the main differences between a local sports club and an international open source project?

   Open source projects are not bound to a specific language or location. Contributors can live on different countries and continents, have different native languages and even live in different timezones. Most of the activity carried out in an open source project happens virtually, not in person.

2. Why can contributing to an open source project be particularly rewarding?

   Many open source projects have a diverse group of people. By collaborating with them, you can learn from them, discover new things, and widen your scope.

3. Which bug tracking software does the Ubuntu project use?

   Launchpad

4. What is the name of the website for the Linux kernel mailing lists?

   `https://lkml.org/`

Imprint

The LPI Learning Materials are an initiative of Linux Professional Institute (https://lpi.org). Learning Materials and their translations can be found at https://learning.lpi.org.

For questions and comments on this edition as well as on the entire project write an email to: learning@lpi.org.