

Open Source Essentials

Version 1.0
Français

050

Table of Contents

THÈME 051: FONDEMENTS DES LOGICIELS	1
051.1 Composants des logiciels	2
Leçon 1	3
Introduction	3
Qu'est-ce qu'un logiciel ?	3
Langages de programmation	4
Paradigmes de programmation	13
Exercices guidés	15
Exercices d'exploration	16
Résumé	17
Réponses aux exercices guidés	18
Réponses aux exercices d'exploration	19
051.2 Architecture logicielle	21
Leçon 1	22
Introduction	22
Serveurs et clients	23
Applications web	24
Interface de programmation d'application (API)	26
Types d'architectures	28
Exercices guidés	32
Exercices d'exploration	33
Résumé	34
Réponses aux exercices guidés	35
Réponses aux exercices d'exploration	37
051.3 Informatique sur site et dans le cloud	38
Leçon 1	39
Introduction	39
Informatique sur site et dans le cloud	39
Modèles communs de fonctionnement du cloud	42
Types communs de services dans le cloud	43
Principaux avantages et risques du cloud et de l'infrastructure informatique sur site	44
Exercices guidés	47
Exercices d'exploration	48
Résumé	49
Réponses aux exercices guidés	50
Réponses aux exercices d'exploration	51
THÈME 052: LICENCES DE LOGICIELS À CODE SOURCE OUVERT	52
052.1 Concepts des licences de logiciels à code source ouvert	53

Leçon 1	55
Introduction	55
Définitions des logiciels open source et des logiciels libres	56
Libre comme la parole : La vraie liberté pour les utilisateurs	56
Logiciels à code source ouvert (<i>open source</i>)	57
Liberté vs Open Source	58
Autres types de logiciels gratuits	58
Principes de la législation sur les droits d'auteur et leur incidence sur les licences des logiciels open source	59
Principes du droit des brevets	60
Contrats de licence	61
Travaux dérivés	62
Conséquences des violations de licence	63
Compatibilité et incompatibilité des licences	63
Double licence et licences multiples	64
Exercices guidés	65
Exercices d'exploration	67
Résumé	68
Réponses aux exercices guidés	69
Réponses aux exercices d'exploration	71
052.2 Licences de logiciels copyleft	72
Leçon 1	74
Introduction	74
Copyleft et la licence publique générale GNU	74
Licence publique générale GNU Affero	79
Compatibilité des licences copyleft	79
Œuvres combinées et dérivées	80
Copyleft faible	81
Exercices guidés	83
Exercices d'exploration	84
Résumé	85
Réponses aux exercices guidés	86
Réponses aux exercices d'exploration	87
052.3 Licences de logiciels permissives	89
Leçon 1	90
Introduction	90
Droits et obligations des licences de logiciel permissives	91
Caractéristiques des principales licences logicielles permissives	91
Licences de logiciels permissives par rapport aux autres licences open source	96
Exercices guidés	98

Exercices d'exploration	100
Summary	101
Réponses aux exercices guidés	102
Réponses aux exercices d'exploration	104
THÈME 053: LICENCES DE CONTENU OUVERT	105
053.1 Concepts des licences de contenu ouvert	106
Leçon 1	108
Introduction	108
Principes fondamentaux du droit d'auteur	109
Caractéristiques communes des licences de contenu ouvert	111
Importance des licences de contenu ouvert	112
Marques déposées et droits d'auteur	113
Exercices guidés	115
Exercices d'exploration	116
Résumé	117
Réponses aux exercices guidés	118
Réponses aux exercices d'exploration	119
053.2 Licence Creative Commons	120
Leçon 1	121
Introduction	121
Origine et objectifs de Creative Commons	122
Modules de la licence Creative Commons	123
Licences Creative Commons de base	125
Creative Commons Zéro (CC0) et la marque du domaine public	128
Sélection de licences et marquage des œuvres	130
Licences internationales et licences adaptées	132
Exercices guidés	133
Exercices d'exploration	134
Résumé	135
Réponses aux exercices guidés	136
Réponses aux exercices d'exploration	137
053.3 Autres licences de contenu ouvert	138
Leçon 1	139
Introduction	139
Licences pour la documentation (logicielle)	139
Licences pour bases de données	141
Libre accès	145
Exercices guidés	148
Exercices d'exploration	149
Résumé	150

Réponses aux exercices guidés	151
Réponses aux exercices d'exploration	152
THÈME 054: MODÈLES D'ENTREPRISE DES LOGICIELS OPEN SOURCE	153
054.1 Modèles d'entreprises de développement logiciel	154
Leçon 1	156
Introduction	156
Objectifs et raisons de publier un logiciel ou un contenu sous une licence libre	157
Modèles d'entreprises courants et sources de revenus	158
Utilisation de logiciels open source dans d'autres technologies et services	160
Considérations relatives aux logiciels open source du point de vue du client	161
Structures de coûts et investissements	162
Exercices guidés	164
Exercices d'exploration	165
Résumé	166
Réponses aux exercices guidés	167
Réponses aux exercices d'exploration	168
054.2 Modèles d'entreprises de fournisseurs de services	169
Leçon 1	171
Introduction	171
Sources de revenus	172
Impact des licences	174
Considérations relatives à la sécurité et à la protection de la vie privée	175
Accords communs entre le fournisseur de services et le client	176
Coûts et investissements	178
Exercices guidés	179
Exercices d'exploration	180
Résumé	181
Réponses aux exercices guidés	182
Réponses aux exercices d'exploration	183
054.3 Conformité et atténuation des risques	184
Leçon 1	186
Introduction	186
Exigences relatives à la publication de logiciels basés sur des composants open source	186
Risques liés aux logiciels open source	189
Nomenclature de logiciels : Savoir ce que vous utilisez	192
Stratégies formelles et conformité	194
Exercices guidés	197
Exercices d'exploration	198
Résumé	199
Réponses aux exercices guidés	200

Réponses aux exercices d'exploration	201
THÈME 055: GESTION DE PROJETS	202
055.1 Modèles de développement de logiciels	203
Leçon 1	204
Introduction	204
Rôles dans le développement de logiciels	204
Planification et ordonnancement	206
Outils courants	206
Modèle en cascade	207
Développement logiciel agile, Scrum et Kanban	209
DevOps	215
Exercices guidés	217
Exercices d'exploration	218
Résumé	219
Réponses aux exercices guidés	220
Réponses aux exercices d'exploration	221
055.2 Gestion des produits / Gestion des versions	222
Leçon 1	224
Introduction	224
Caractéristiques des versions	224
Versions logicielles : majeures, mineures et correctifs	227
Cycle de vie d'un logiciel	228
Documentation pour les versions du produit	230
Exercices guidés	231
Exercices d'exploration	232
Résumé	233
Réponses aux exercices guidés	234
Réponses aux exercices d'exploration	235
055.3 Gestion des communautés	236
Leçon 1	238
Introduction	238
Rôles dans les projets open source	238
Tâches courantes dans les projets open source	240
Types de contributions dans l'open source	242
Types de contributeurs dans l'open source	242
Rôle des organisations dans les projets open source	243
Transfert de droits	245
Règles et stratégies	245
Attribution et transparence	246
Diversité, équité, inclusion et non-discrimination	247

Exercices guidés	249
Exercices d'exploration	250
Résumé	251
Réponses aux exercices guidés	252
Réponses aux exercices d'exploration	253
THÈME 056: COLLABORATION ET COMMUNICATION	254
056.1 Outils de développement	255
Leçon 1	257
Introduction	257
Objectifs de développement	257
Processus généraux de développement	258
Outils usuels de développement logiciel	262
Types de test logiciels courants	266
Environnements de déploiement courants	268
Exercices guidés	270
Exercices d'exploration	271
Résumé	272
Réponses aux exercices guidés	273
Réponses aux exercices d'exploration	274
056.2 Gestion du code source	275
Leçon 1	276
Introduction	276
Système de gestion du code source et dépôt	277
Commits, Tags, and Branches	278
Sous-dépôts	280
Utilisation générale d'un système de gestion du code source	280
Systèmes de gestion de versions courants	281
Exercices guidés	283
Exercices d'exploration	284
Résumé	285
Réponses aux exercices guidés	286
Réponses aux exercices d'exploration	287
056.3 Outils de communication et de collaboration	288
Leçon 1	290
Introduction	290
Modes de communication	291
Outils de communication	293
Outils de collaboration	298
Gestion du code source	301
Documentation	302

Exercices guidés	304
Exercices d'exploration	305
Résumé	306
Réponses aux exercices guidés	307
Réponses aux exercices d'exploration	309
Impression	310



**Linux
Professional
Institute**

Thème 051: Fondements des logiciels



Linux
Professional
Institute

051.1 Composants des logiciels

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 051.1

Valeur

2

Domaines de connaissance les plus importants

- Comprendre le concept de code source et d'exécution du code
- Comprendre le concept de compilateurs et d'interpréteurs
- Comprendre le concept de bibliothèques logicielles

Liste partielle de termes, fichiers et utilitaires utilisés

- Code source
- Programmes exécutables
- Code à octets (Bytecode)
- Code machine
- Compilateur
- Éditeur de liens (Linker)
- Interpréteur
- Machine virtuelle d'exécution
- Algorithme
- Bibliothèques logicielles
- Liaison statique et dynamique



Linux
Professional
Institute

Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	051 Fondements des logiciels
Objectif :	051.1 Composants des logiciels
Leçon :	1 sur 1

Introduction

Les logiciels *libres* et à *code source ouvert*—souvent abrégés en FOSS (*Free and Open-Source Software*)—font désormais partie intégrante de notre vie quotidienne, la plupart du temps sans que nous en soyons conscients. Les FOSS, par exemple, se retrouvent sous une forme ou une autre derrière toutes nos activités sur l'internet : sur l'ordinateur sur lequel nous visualisons les pages web dans le navigateur, ou sur les serveurs qui stockent ces pages web et les livrent dès que nous les appelons.

Qu'est-ce qu'un logiciel ?

Toutefois, avant d'examiner toutes les spécificités des logiciels libres et à code source ouvert, il convient de préciser ce qu'est un *logiciel*. Commençons par une description très générale : Le logiciel est la partie non physique et immatérielle des ordinateurs, quelle que soit leur forme. Le logiciel garantit que les parties physiques (le *matériel*) de l'ordinateur interagissent et que l'ordinateur peut accepter des commandes et exécuter des tâches.

Un smartphone, un ordinateur portable ou un serveur dans un centre de données n'est qu'une machine faite de métal et de plastique lorsqu'il est éteint. Dès qu'il est allumé, un logiciel démarre,

sous la forme de séquences de commandes codées qui contrôlent les différents composants de cette machine, qui permettent à l'utilisateur d'interagir avec la machine et qui exécutent des tâches très spécifiques en invoquant des applications individuelles.

C'est le travail des *développeurs de logiciels* d'analyser les tâches que l'ordinateur est censé accomplir et de les spécifier d'une manière qui permette à l'ordinateur de les mettre en œuvre. Les outils utilisés par les développeurs sont aussi nombreux et divers que les tâches accomplies par le logiciel.

Certaines tâches logicielles sont très étroitement liées au matériel et à l'architecture de l'ordinateur, par exemple l'adressage et la gestion de la mémoire ou la gestion de différents processus. Les *programmeurs système* travaillent donc à un niveau très proche du matériel.

Les *développeurs d'applications*, quant à eux, se concentrent davantage sur l'utilisateur et programment des applications qui permettent aux utilisateurs d'effectuer leurs tâches de manière efficace et intuitive. Un exemple d'application complexe est un programme de traitement de texte qui fournit toutes les fonctions de mise en forme du texte dans des menus ou des boutons et qui affiche également le texte tel qu'il pourrait être imprimé.

En général, un *algorithme* est une façon de résoudre un problème. Par exemple, pour calculer une moyenne, l'algorithme normal consiste à additionner une série de valeurs et à diviser la somme par le nombre total de valeurs. Bien que les algorithmes soient traditionnellement conçus et exécutés par des programmeurs, ils sont également générés de nos jours par l'intelligence artificielle.

Les concepts présentés dans ce chapitre peuvent vous aider à comprendre les avantages et les risques des logiciels libres, à faire des choix éclairés et même à décider si vous voulez devenir un développeur logiciel.

Langages de programmation

Les *langages de programmation* sont des langages artificiels hautement structurés qui indiquent à un ordinateur ce qu'il doit faire. Les programmes sont généralement écrits sous forme de texte, mais certains langages sont écrits sous forme graphique. Les développeurs logiciels écrivent des instructions (appelées *code*) à l'ordinateur dans ce langage artificiel. Cependant, le matériel de l'ordinateur n'exécute pas directement ce code. Le matériel ne peut exécuter directement qu'une suite de séquences de bits stockés dans la mémoire, appelée *code machine* ou *langage machine*. Tous les langages de programmation sont soit convertis en code machine par un *compilateur*, soit interprétés par un autre programme en code machine appelé *interpréteur* pour que le matériel exécute ces instructions.

Parmi les langages de programmation les plus utilisés actuellement figurent Python, JavaScript, C, C++, Java, C#, Swift et PHP. Chacun de ces langages de programmation a ses propres forces et faiblesses, et le choix du langage dépend du projet et des besoins du développeur. Par exemple, Java est un choix populaire pour le développement d'applications destinées aux très grandes entreprises, tandis que Python est souvent utilisé pour le calcul scientifique et l'analyse de données.

Les développeurs ont fait preuve d'une créativité impressionnante dans la conception des langages de programmation. À l'origine, il s'agissait de *langages de bas niveau* qui ressemblaient aux instructions de l'ordinateur. Les langages sont devenus de plus en plus de *haut niveau*, c'est-à-dire qu'ils tentent de représenter de puissantes combinaisons d'instructions en termes brefs. Certains langages reflètent la façon dont les gens pensent naturellement, tout en préservant la rigueur nécessaire pour fonctionner correctement.

Environ 400 langages de programmation sont actuellement connus, bien que nombre d'entre eux ne soient utilisés que dans des applications très spécifiques ou dans des anciens systèmes. Chacun d'entre eux a été développé pour résoudre certaines tâches.

Caractéristiques, syntaxe et structure des langages de programmation

Le choix du langage de programmation peut avoir un impact significatif sur les performances, l'extensibilité et la facilité de développement d'un projet logiciel. Ces sections présentent les éléments importants des langages.

Caractéristiques des langages de programmation

Parmi les caractéristiques et qualités communes des langages de programmation il y a :

Concurrence

La concurrence désigne le traitement simultané de plusieurs tâches, soit en les exécutant sur différents processeurs matériels, soit en alternant l'utilisation d'un même processeur par les tâches. Le degré de concurrence pris en charge par un langage de programmation peut affecter considérablement ses performances et son extensibilité, en particulier pour les applications qui nécessitent un traitement en temps réel ou de grandes quantités de données. Chaque tâche distincte peut être appelée un *processus*, une *tâche* ou un *thread*.

Gestion de la mémoire

La gestion de la mémoire est l'allocation et la libération de la mémoire dans un programme. En fonction du langage ou de l'environnement d'exécution, la gestion de la mémoire peut être effectuée manuellement par le programmeur ou gérée automatiquement. Une bonne gestion de la mémoire est essentielle pour garantir qu'un programme utilise efficacement la mémoire

et qu'il n'en manque pas ou ne cause pas d'autres problèmes. Si un programme ne parvient pas à libérer la mémoire inutilisée, il provoque une *fuite de mémoire* qui augmente progressivement l'utilisation de la mémoire jusqu'à ce que le programme se bloque ou que des effets négatifs sur les performances soient perceptibles.

Mémoire partagée

La mémoire partagée est un type de mécanisme de communication inter-processus qui permet à plusieurs processus de lire et de manipuler une région commune de la mémoire. La mémoire partagée est courante dans le matériel tel que les lecteurs de disques et peut également être un moyen efficace de partager des données entre les processus. Toutefois, ce mécanisme nécessite une synchronisation et une gestion minutieuse afin d'éviter la corruption des données. Une erreur connue sous le nom de *course critique* (*race condition*) se produit si un processus apporte une modification imprévue à des données alors qu'un autre processus les utilise.

Transmission de messages

La transmission de messages est un mécanisme de communication entre processus qui leur permet d'échanger des données et de coordonner leurs activités. Il est couramment utilisé dans la programmation concurrente pour assurer la communication inter-processus et peut être mis en œuvre par divers mécanismes tels que les *sockets*, les tubes (*pipes*) ou les files d'attente de messages.

Ramasse-miettes

Le ramasse-miettes (*Garbage collection*) est une technique de gestion automatique de la mémoire utilisée par certains langages de programmation pour récupérer la mémoire qui n'est plus utilisée pendant l'exécution d'un processus. Cette technique permet d'éviter les fuites de mémoire et facilite l'écriture d'un code correct et efficace par les développeurs, mais elle peut également introduire des surcoûts de performance et rendre plus difficile le contrôle du comportement précis du programme.

Types de données

Les types de données déterminent le type d'informations qui peuvent être représentées dans le programme. Les types de données peuvent être prédéfinis dans le langage ou définis par l'utilisateur, et peuvent inclure des nombres entiers, des nombres à virgule flottante (c'est-à-dire des approximations de nombres réels), des chaînes de caractères, des tableaux et d'autres.

Entrée et sortie (E/S)

Les entrées et les sorties sont des mécanismes permettant de lire et d'écrire des données vers et depuis un programme. Les entrées peuvent provenir de diverses sources, telles que les clics de l'utilisateur et les saisies au clavier, un fichier ou une connexion réseau, tandis que les sorties peuvent être envoyées vers diverses destinations, telles qu'un écran, un fichier ou une

connexion réseau. Les E/S permettent aux programmes d'interagir avec le monde extérieur et d'échanger des informations avec d'autres systèmes.

Gestion des erreurs

La gestion des erreurs détecte et répond aux erreurs qui se produisent pendant l'exécution d'un programme. Il s'agit notamment d'erreurs telles que la division par zéro ou le fait qu'un fichier demandé soit introuvable. La gestion des erreurs permet aux programmes de continuer à fonctionner même lorsque des erreurs se produisent, améliorant ainsi leur fiabilité et leur robustesse.

Les concepts énumérés ci-dessus sont fondamentaux pour comprendre le fonctionnement des langages de programmation et la manière d'écrire un code efficace et facile à maintenir.

Syntaxe des langages de programmation

La *syntaxe* d'un langage de programmation fait référence aux règles d'écriture des déclarations et des expressions du programme. Il est important que la syntaxe soit *bien définie* et *consistante*, afin que le programmeur puisse écrire et comprendre efficacement son code. Les éléments suivants sont les composantes de base de la plupart des langages de programmation :

Procédures et fonctions

Les procédures et les fonctions sont utilisées pour définir des blocs de code réutilisables qui peuvent être appelés plusieurs fois.

Variables

Les variables représentent des morceaux de mémoire et stockent des données qui peuvent être manipulées et transmises entre les procédures et les fonctions.

Opérateurs

Les opérateurs sont des mots-clés ou des symboles (tels que `+` et `-`) qui attribuent des valeurs aux variables et effectuent des opérations arithmétiques.

Structure de contrôle

En général, le code du programme est exécuté dans l'ordre dans lequel il est écrit, mais les *instructions conditionnelles* modifient le flux d'exécution. Le code qui est exécuté ensuite dépend de diverses conditions, telles que le contenu de la mémoire, l'état du clavier, les paquets arrivant du réseau, etc. Les *instructions itératives*, une forme spéciale d'instructions conditionnelles, sont utiles pour effectuer les mêmes opérations sur une série d'ensembles de données. Une *exception*, qui invoque un code spécial en cas d'erreur, est une autre structure de contrôle.

La syntaxe et le comportement de ces constructions peuvent varier d'un langage de programmation à un autre, et le choix du langage peut avoir un impact important sur la lisibilité et la facilité de maintenance du code.

Bibliothèques

Un bon langage de programmation doit permettre de développer facilement des programmes et de réutiliser facilement le code existant. De nombreux langages de programmation disposent d'un mécanisme permettant d'organiser les procédures et les fonctions en parties réutilisables dans d'autres programmes.

Une *bibliothèque* est un ensemble de procédures et de fonctions destinées à supporter une caractéristique ou un objectif particulier, regroupées dans un seul fichier. La disponibilité de nombreuses bibliothèques faciles à utiliser est une autre exigence très importante d'un bon langage de programmation. Par exemple, Python est largement reconnu comme un bon langage pour développer des programmes liés à l'intelligence artificielle, car il dispose d'un certain nombre de bibliothèques adaptées au traitement par l'intelligence artificielle.

Avec l'augmentation de la taille et de la complexité des programmes, les bibliothèques en tant que blocs de construction prêts à l'emploi deviennent de plus en plus importantes. C'est particulièrement vrai dans le monde de l'open source, où les utilisateurs sont à l'aise avec le code créé par d'autres et le réutilisent. En conséquence, un écosystème de bibliothèques s'est développé pour chaque langage de programmation, et les gestionnaires de paquets tels que `composer` pour PHP, `pip` pour Python, et `gems` pour Ruby facilitent l'installation des bibliothèques.

Les bibliothèques sont également importantes dans les langages compilés. La combinaison de plusieurs fichiers binaires et de bibliothèques pré-compilées pour obtenir un seul fichier exécutable est appelée *édition de liens*, et l'outil qui effectue cette opération est appelé *éditeur de liens*. Il existe deux types d'édition de liens : la *liaison statique*, dans laquelle seul le code de la bibliothèque nécessaire est inclus dans le fichier exécutable de l'application finale et la *liaison dynamique*, dans laquelle une bibliothèque installée dans le système est partagée par toutes les applications qui utilisent cette bibliothèque. Actuellement, la liaison dynamique est l'approche préférée et se caractérise par des exécutables d'application plus petits et une utilisation moindre de la mémoire au moment de l'exécution.

Notez que, comme les mêmes bibliothèques peuvent être utilisées par plusieurs programmes, les différences entre les versions d'une bibliothèque peuvent être encore plus problématiques que pour les applications. Faisons une parenthèse et rappelons-nous comment regarder les numéros de version. La *gestion sémantique de version* (*semantic versioning*) est couramment utilisée, qui indique les versions par trois nombres séparés par des points. Une version typique peut-être 2.39.16, qui indique une version majeure de 2 (un nombre qui ne changera probablement

qu'une fois toutes les quelques années), une version mineure de 39 dans la version majeure (qui peut être mise à jour tous les quelques mois pour contenir des changements de fonctionnalités importants), et une révision rapide de 16 (qui peut changer à cause d'une seule correction de bogue). Les versions et révisions ultérieures ont des numéros plus élevés.

Un exemple très simple

Examinons un exemple *très* simple de programme informatique en langage Python pour avoir une idée sommaire de quelques-uns des éléments mentionnés.

Exprimé en langage naturel, le programme est censé faire ce qui suit : « Demander à l'utilisateur d'entrer un nombre et vérifier si ce nombre est pair ou impair. Enfin, affichez le résultat. »

Et voici le code que nous pouvons sauvegarder dans le fichier `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.")
```

Même dans ces quelques lignes de code, on retrouve un grand nombre des caractéristiques et des éléments de syntaxe mentionnés ci-dessus :

1. À la ligne 1, nous définissons la *variable* `num` et lui assignons une *valeur* avec l'_opérateur `=`.
2. La valeur attribuée correspond à l'*entrée* de l'utilisateur (via la *fonction* `input()`). En outre, la fonction `int()` veille à ce que cette entrée soit convertie, si c'est possible, en un *type de données* entier. L'expression passée à une fonction entre parenthèses est appelée *paramètre* ou *argument*.
3. Si l'utilisateur entre une chaîne de caractères, Python affichera une erreur dans le cadre de sa *gestion des erreurs*. Si un nombre décimal est saisi, la fonction `int()` le convertit en un nombre entier, par exemple `5.73` en `5`.
4. Dans les lignes suivantes, la *structure de contrôle* énonçant une *condition*, avec les mots-clés `if` et `else`, contrôle ce qui se passe dans chacun des deux cas possibles (le nombre est soit pair, soit impair).
5. Tout d'abord, l'opérateur modulo `%` teste si (`if`) la division du nombre saisi par 2 donne la valeur 0 (c'est-à-dire pas de reste) — dans ce cas, le nombre est pair. L'opérateur doublé `==` est l'opérateur de comparaison « est égal à », qui est différent de l'opérateur d'affectation `=` de la ligne 1.

6. Dans l'autre cas (`else`), c'est-à-dire lorsque la division par 2 donne un résultat différent de 0, le nombre saisi doit être impair.
7. Dans les deux cas, la fonction `print()` renvoie le résultat en tant que *sortie* sous forme textuelle.

Et voici à quoi cela ressemble lorsque nous exécutons le programme sur la ligne de commande :

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

Lorsque vous considérez la quantité de logique du langage déjà présente dans ce petit exemple, vous pouvez vous faire une idée de ce dont sont capables les logiciels complexes distribués sur des milliers de fichiers ; par exemple, les *systèmes d'exploitation* tels que Microsoft Windows, macOS ou Linux, qui rendent tout le matériel d'un ordinateur disponible et garantissent en même temps que les utilisateurs peuvent installer toutes les autres applications souhaitées et les utiliser pour le travail ou pour le plaisir.

Code machine, langage d'assemblage et assembleurs

Comme indiqué précédemment, le matériel ne peut exécuter directement qu'une suite de séquences de bits appelées code machine. Le processeur (*CPU*) lit une séquence de bits de la mémoire en unités d'un mot (8 à 64 bits) et exécute l'instruction correspondante. Les instructions individuelles sont assez simples, par exemple « copier le contenu de l'emplacement mémoire A vers l'emplacement mémoire B », « multiplier le contenu de l'emplacement mémoire C par le contenu de l'emplacement mémoire D » ou « lire les données qui sont arrivées au périphérique à l'adresse X ». De nos jours, le nombre de modèles d'instructions a augmenté de façon exponentielle et il n'est plus possible de se souvenir de tous ces modèles.

Le code machine est une suite de séquences de bits, ou des 0 et des 1, ce qui n'est pas du tout intuitif pour les êtres humains. Pour rendre la programmation plus intuitive, le *langage d'assemblage* a été créé, dans lequel les instructions reçoivent des noms et peuvent être spécifiées par des chaînes de caractères. Dans le langage d'assemblage, les instructions qui correspondent une à une au code machine sont écrites une à la fois. Les instructions peuvent ressembler à ce qui suit :

<code>move</code>	<code>[B], [A]</code>	copier le contenu de la mémoire A vers la mémoire B
<code>multi</code>	<code>R1, [C], [D]</code>	multiplier le contenu de la mémoire C par celui de la mémoire D
<code>input</code>	<code>R1, [X]</code>	lire les données qui sont arrivées au périphérique à l'adresse X

Une instruction en langage d'assemblage correspond à une instruction en code machine, qui correspond à l'instruction exacte que le matériel peut comprendre et exécuter. Les avantages du langage d'assemblage par rapport au langage machine sont les suivants :

Amélioration de la lisibilité et de la maintenabilité

Le langage d'assemblage est beaucoup plus facile à lire et à écrire que le code machine. Il est donc plus simple pour les programmeurs de comprendre, de déboguer et de maintenir leur code.

Automatisation du calcul des adresses

La programmation en code machine peut également utiliser le concept de variables et de fonctions, mais tout doit être exprimé en termes d'*adresses mémoire*. Le langage d'assemblage attribue aussi des noms aux adresses mémoire, ce qui facilite la formulation de la logique d'un programme.

Puisque le langage d'assemblage a accès à toutes les fonctionnalités du matériel, il est couramment utilisé dans les situations suivantes :

Partie du système d'exploitation qui dépend de l'architecture

L'utilisation d'instructions spécifiques à une architecture de processeur, pour l'accès aux fonctions d'initialisation et de sécurité, ne peut se faire qu'en langage d'assemblage.

Développement de composants système de bas niveau

Le langage d'assemblage est utilisé pour développer les composants du système qui doivent interagir directement avec le matériel de l'ordinateur, tels que les pilotes de périphériques, les micrologiciels (*firmware*) et le système élémentaire d'entrée/sortie (BIOS - *Basic Input/Output System*). En particulier, les dispositifs à grande vitesse qui nécessitent de pousser les performances matérielles à leurs limites ont souvent besoin de pilotes et de micrologiciels programmés en langage d'assemblage.

Programmation des microcontrôleurs

Le langage d'assemblage est également utilisé pour programmer les microcontrôleurs, qui sont de petits ordinateurs de faible puissance utilisés dans un large éventail de systèmes intégrés, des jouets au contrôle industriel. Certains microcontrôleurs ont une capacité de mémoire de quelques centaines d'octets seulement et sont généralement programmés en langage d'assemblage.

Le code du langage d'assemblage est converti en code machine avant d'être exécuté par une application appelée *assembleur*. L'assembleur est le plus ancien outil de programmation et a apporté un certain nombre d'avantages qui étaient impensables dans la programmation en code machine. Pour brouiller les pistes, le langage d'assemblage est parfois appelé assembleur.

Le code machine et le langage d'assemblage diffèrent d'un processeur à un autre. Ils sont appelés « langages de bas niveau » parce qu'ils opèrent directement sur le matériel. Cependant, les concepts de calcul et d'entrée/sortie sont les mêmes pour tous les processeurs. Si les concepts communs peuvent être exprimés d'une manière plus facile à comprendre pour les êtres humains, l'efficacité de la programmation peut être considérablement améliorée. C'est là qu'interviennent les « langages de haut niveau ».

Langages compilés

Les *langages compilés* sont des langages de programmation qui sont traduits soit en code machine, soit dans un format intermédiaire appelé *bytecode* (code à octets). Le *bytecode* est exécuté sur l'ordinateur cible par une *machine virtuelle d'exécution*. La machine virtuelle traduit le *bytecode* en code machine adapté à chaque ordinateur. Le *bytecode* permet aux programmes d'être indépendants de la plate-forme et de fonctionner sur n'importe quel système doté d'une machine virtuelle compatible.

La traduction du code source écrit dans un langage de programmation de haut niveau en code machine ou en *bytecode* est effectuée par un *compilateur*. Parmi les exemples de langages compilés qui produisent directement du code machine, on peut citer C et C++. Les langages qui produisent du *bytecode* incluent Java et C#.

Le choix entre le code machine et le *bytecode* dépend des exigences du projet, telles que la performance, la neutralité par rapport à la plate-forme et la facilité de développement.

Langages interprétés

Les *langages interprétés* sont des langages de programmation qui sont exécutés par un *interpréteur*, plutôt que d'être compilés en code machine. L'interpréteur lit le code source et exécute les instructions qu'il contient. Un interpréteur est capable de traiter directement le code source, sans le transformer dans un autre format de fichier. Ainsi, contrairement à un compilateur, qui traduit l'ensemble du programme en code machine avant de l'exécuter, un interpréteur lit chaque ligne de code et l'exécute immédiatement, ce qui permet au programmeur de voir les résultats de chaque ligne au fur et à mesure qu'elle est exécutée.

Les langages interprétés sont couramment utilisés pour écrire des *scripts*, c'est-à-dire des programmes courts qui automatisent des tâches, pour les interfaces en ligne de commande et pour le contrôle des lots et des travaux. Les scripts écrits dans des langages interprétés peuvent être facilement modifiés et exécutés sans nécessiter une recompilation, ce qui les rend bien adaptés aux tâches qui nécessitent un prototypage rapide ou une itération souple et rapide. Cette commodité s'accompagne de quelques inconvénients potentiels. Par exemple, un programme interprété s'exécute plus lentement que son équivalent compilé.

Python, Ruby et JavaScript sont des exemples de langages interprétés. Python est largement utilisé dans l'informatique scientifique, l'analyse de données et l'apprentissage automatique, tandis que Ruby est souvent utilisé dans le développement web et pour créer des scripts d'automatisation. JavaScript est un langage de script côté client intégré dans les navigateurs web pour créer des pages web dynamiques et interactives.

Langages orientés données

Les *langages orientés données* sont des langages de programmation optimisés pour le traitement et la manipulation de grandes quantités de données. Ils sont conçus pour traiter efficacement de grands ensembles de données structurées ou non structurées et fournissent un ensemble d'outils pour travailler avec des bases de données, des structures de données et des algorithmes pour le traitement et l'analyse des données.

Les langages orientés données sont utilisés dans une variété d'applications, y compris les sciences des données, l'analyse des données massives (*big data*), l'apprentissage automatique (*machine learning*) et la programmation des bases de données. Ils sont bien adaptés aux tâches qui impliquent le traitement et l'analyse de grandes quantités de données, telles que le nettoyage et la transformation des données, la visualisation des données et la modélisation statistique.

Parmi les exemples de langages orientés données, il y a SQL (abréviation de *Structured Query Language* ou langage de requêtes structurées), R et MATLAB. SQL est un langage standard utilisé pour gérer les bases de données relationnelles et il est largement utilisé dans les entreprises et l'industrie. R est un langage de programmation et un environnement pour le calcul statistique et les graphiques. Il est largement utilisé dans la science des données et l'apprentissage automatique. MATLAB est un environnement de calcul numérique et un langage de programmation utilisé dans un large éventail d'applications, notamment le traitement des signaux, le traitement des images et la finance informatisée.

Paradigmes de programmation

Outre les spécificités des langages de programmation, les *paradigmes de programmation* déterminent l'approche particulière à adopter pour trouver une solution. Nous pouvons considérer un paradigme comme une stratégie de base avec laquelle nous abordons une tâche, en fonction des exigences et des conditions spécifiques.

Un exemple comparable est la construction d'une maison : Que les maçons montent les murs brique par brique ou que les éléments en béton préfabriqués soient assemblés sur place est une décision fondamentale qui dépend des exigences et des circonstances. Quelles sont les caractéristiques souhaitées pour la maison ? Où est-elle située ? Est-elle reliée à d'autres maisons ?

De même, les paradigmes définissent l'orientation de la programmation : comment et de quelle manière, par exemple, un projet logiciel est divisé en parties plus petites et distinctes. Chaque langage de programmation convient mieux à un paradigme particulier. Le choix du paradigme est donc étroitement lié au choix du langage de programmation.

Les paradigmes suivants sont répandus dans la programmation :

Programmation orientée objet (POO)

La POO est basée sur le concept d'*objets*, qui sont des instances de *classes* qui encapsulent les données et le comportement. Par exemple, un langage peut proposer un rectangle comme classe pour aider le programmeur à afficher une boîte à l'écran.

La POO se concentre sur la manipulation des données au niveau de l'objet. La POO facilite l'écriture d'un code qui est maintenable, réutilisable et extensible. Elle est largement utilisée dans les logiciels de bureau, les jeux vidéo et les applications web. Java, C# et Python sont des exemples de langages de programmation orientés objet.

Programmation procédurale

La programmation procédurale exécute des tâches via des *procédures*, ou des blocs de code qui peuvent être exécutés dans un ordre spécifique. Cela permet d'écrire un code structuré, facile à suivre, mais peut conduire à un code moins flexible et plus difficile à maintenir à mesure que la taille et la complexité du projet augmentent. Le C, le Pascal et le Fortran sont des exemples de langages de programmation procéduraux.

D'autres approches du développement de logiciels sont utilisées aujourd'hui, pour lesquelles certains langages sont mieux adaptés que d'autres. En outre, les interfaces glisser-déposer (*drag-and-drop*) permettent aux non-programmeurs d'écrire des programmes, et de nombreux services en ligne ont récemment commencé à générer du code grâce à l'intelligence artificielle lorsqu'on leur donne un texte en langage courant.

En conclusion, chaque paradigme de programmation a ses propres forces et faiblesses, et le choix du paradigme dépend souvent des besoins du projet, de l'expérience et des préférences du développeur, ainsi que des contraintes de la plateforme et de l'environnement de développement. Comprendre les différents types de paradigmes peut vous aider à choisir le bon paradigme pour vos besoins, et peut également vous aider à écrire un code meilleur et plus efficace.

Exercices guidés

1. Quel est l'objectif des fonctions ?

2. Quel est l'avantage du *bytecode* par rapport à un fichier de code machine ?

3. Quel est l'avantage d'un fichier en code machine comparé à un fichier en *bytecode* ?

Exercices d'exploration

1. Quels sont les inconvénients de la décomposition d'un programme en un grand nombre de processus ou de tâches ?

2. Vous avez trouvé plusieurs paquets open source, proposés dans différentes versions, qui offrent les fonctionnalités dont vous avez besoin pour votre programme. Quels sont les critères de choix d'un paquet ?

3. Outre les paradigmes de développement POO et procédural, quelles sont les autres approches de développement logiciel existantes et quel est le langage de programmation le mieux adapté à chacune d'entre elles ?

Résumé

Dans cette leçon, vous avez appris ce qu'est un logiciel et comment il est développé à l'aide de langages de programmation. Les nombreux langages de programmation se distinguent non seulement par leur syntaxe, mais aussi, par exemple, par leur gestion des ressources matérielles ou des structures de données.

Les langages de programmation diffèrent également par la manière dont le code source, lisible par les êtres humains, est converti par un interpréteur ou un compilateur en code machine final à exécuter par l'ordinateur.

Les paradigmes de programmation déterminent la stratégie des projets logiciels et donc le choix des langages de programmation appropriés, en fonction des exigences et de la taille du projet concerné.

Réponses aux exercices guidés

1. Quel est l'objectif des fonctions ?

Les fonctions encapsulent certaines activités courantes, telles que l'édition d'une chaîne de caractères. En créant une fonction, vous permettez à votre programme et à d'autres programmes d'exécuter la fonction de manière pratique et répétée sans avoir à écrire leur propre code.

2. Quel est l'avantage du *bytecode* par rapport à un fichier de code machine ?

Le fichier *bytecode* peut être exécuté sur de nombreux ordinateurs différents, où une machine virtuelle transforme le code en code machine. Par exemple, JavaScript fonctionne dans de nombreux navigateurs sur de nombreux types d'ordinateurs.

3. Quel est l'avantage d'un fichier en code machine comparé à un fichier en *bytecode* ?

Le code machine s'exécute le plus rapidement que possible. Le *bytecode* s'exécute plus lentement, car la machine virtuelle doit le transformer en code machine pendant son exécution.

Réponses aux exercices d'exploration

1. Quels sont les inconvénients de la décomposition d'un programme en un grand nombre de processus ou de tâches ?

Lorsqu'un programme est décomposé en processus, ceux-ci doivent communiquer entre eux. S'ils travaillent sur un grand nombre de données communes, les processus risquent de dépenser beaucoup de ressources pour l'échange de données et la protection des données contre des modifications multiples et simultanées (course critique). Les processus entraînent également des surcharges lorsqu'ils démarrent et se terminent. Plus il y a de processus, plus le programme et ses interactions deviennent complexes, de sorte que les erreurs peuvent être plus difficiles à trouver.

Le paradigme fonctionnel tend à faciliter la décomposition des programmes en de nombreux processus, en raison de l'immuabilité. Les données immuables ne souffrent pas de conditions de course.

2. Vous avez trouvé plusieurs paquets open source, proposés dans différentes versions, qui offrent les fonctionnalités dont vous avez besoin pour votre programme. Quels sont les critères de choix d'un paquet ?

Vérifiez les rapports de bogues et les avis de sécurité concernant les paquets, car certains sont très bogués, voire dangereux. Parfois, la dernière version n'est pas la meilleure, parce qu'une faille de sécurité peut y avoir été introduite.

Consultez le forum sur lequel les développeurs parlent du paquet, pour vérifier qu'il est activement maintenu. Votre programme sera probablement utilisé pendant longtemps, et vous voulez que le paquet soit également disponible et robuste au fil du temps.

Essayez différents paquets pour vérifier leurs performances et leur exactitude.

La plupart des paquets dépendent de fonctions trouvées dans d'autres paquets (*dépendances*), de sorte qu'une faiblesse dans l'une des dépendances peut affecter votre programme.

3. Outre les paradigmes de développement POO et procédural, quelles sont les autres approches de développement logiciel existantes et quel est le langage de programmation le mieux adapté à chacune d'entre elles ?

Outre les paradigmes de développement POO et procédural, les autres types de paradigmes sont les suivants.

La programmation fonctionnelle met l'accent sur l'utilisation de fonctions et de concepts

mathématiques, tels que les lambdas et les clôtures (*closures*), pour écrire un code basé sur l'évaluation d'expressions plutôt que sur l'exécution de déclarations. La programmation fonctionnelle traite les fonctions comme des citoyens de première classe — de sorte qu'elles peuvent être manipulées par le programme — et met l'accent sur l'*immuabilité*, ou le fait de travailler avec des variables qui ne peuvent pas changer après avoir été initialement définies. Cela facilite le raisonnement et le test du code, ainsi que l'écriture d'applications concurrentes et parallèles. Erlang, Haskell, Lisp et Scheme sont des exemples de langages de programmation fonctionnels.

Les langages impératifs se concentrent sur les déclarations nécessaires pour contrôler le flux des transitions du programme depuis et vers différents états.

Les langages déclaratifs décrivent les actions à entreprendre et la logique qui se cache derrière les instructions. L'ordre dans lequel les instructions sont exécutées n'est pas spécifié. Ces instructions, appels de fonctions et autres déclarations peuvent être réorganisés et optimisés par les compilateurs tant qu'ils préservent la logique sous-jacente.

La programmation naturelle est un paradigme de programmation qui utilise le langage naturel ou d'autres représentations adaptées à un être humain pour décrire le comportement souhaité d'un programme. L'idée est de rendre la programmation accessible aux personnes qui n'ont pas de formation théorique en informatique. Scratch et Alice sont des exemples de langages de programmation naturelle.



**Linux
Professional
Institute**

051.2 Architecture logicielle

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 051.2

Valeur

2

Domaines de connaissance les plus importants

- Comprendre les concepts de client et de serveur informatique
- Comprendre les concepts de clients légers et de clients lourds
- Comprendre les concepts de monolithes et de microservices et leurs principales différences
- Comprendre les concepts d'interfaces de programmation d'applications (API)
- Comprendre le concept de composants logiciels et leur intégration ou séparation (services, modules, API)

Liste partielle de termes, fichiers et utilitaires utilisés

- Clients et serveurs
- Clients légers et clients lourds
- Applications web
- Applications monopage
- Architectures monolithiques
- Architectures microservices
- Interfaces de programmation d'applications (API)
- API RESTful



Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	051 Fondements des logiciels
Objectif :	051.2 Architecture logicielle
Leçon :	1 sur 1

Introduction

L'internet est omniprésent dans notre monde moderne, tout comme les applications mobiles et web. Ces outils sont utilisés de manière transparente par une grande partie de la population mondiale et permettent de tout faire, de la messagerie instantanée à des activités plus complexes telles que l'achat d'équipements miniers pour de grandes entreprises.

Derrière toutes ces interfaces et tous ces services en ligne apparemment simples se cache une *architecture* — c'est-à-dire un ensemble de logiciels qui coopèrent entre eux — que nous tenons souvent pour acquis. Il faut en savoir un peu plus sur cette architecture pour comprendre comment toutes les pièces de l'internet s'assemblent et comment les logiciels peuvent apporter une réelle valeur ajoutée à leurs utilisateurs.

Dans cette leçon, nous examinerons certaines des architectures logicielles qui se cachent derrière les applications web, qui sont des systèmes logiciels fonctionnant sur des serveurs, et la manière

dont elles sont utilisées dans des systèmes que presque tout le monde connaît.

Serveurs et clients

Lorsque vous utilisez des systèmes en ligne, il est très probable que vous ayez rencontré à un moment ou à un autre un message comme celui de l'Exemple d'écran affiché pendant qu'une réponse est en préparation.

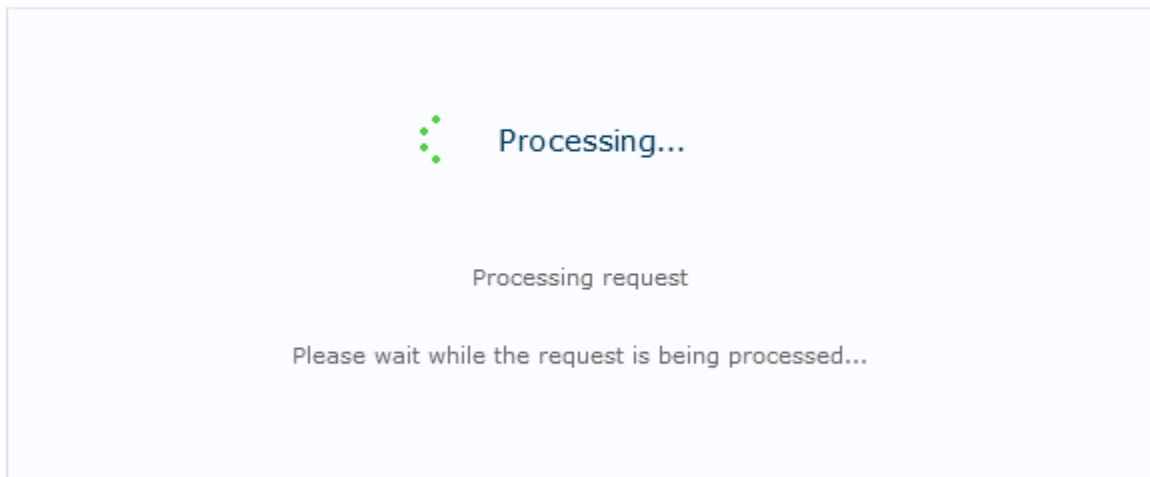


Figure 1. Exemple d'écran affiché pendant qu'une réponse est en préparation

Prenons un peu de recul et examinons le contexte dans lequel vous voyez ce message. Supposons que vous essayez d'accéder à votre compte bancaire par l'intermédiaire du site web de votre banque. Lorsque vous essayez d'accéder au site web de votre banque sur votre ordinateur portable, vous utilisez un type de logiciel (c'est-à-dire une application) appelé navigateur web, tel que Google Chrome ou Firefox. Dans ce cas, le navigateur de votre ordinateur envoie une demande à un autre ordinateur qui héberge le site web. Ce type d'ordinateur est appelé *serveur*. Il est spécialement conçu pour fonctionner 24 heures sur 24, 7 jours sur 7, et répondre en permanence aux nouvelles demandes provenant de toutes les régions du monde.

Un serveur est donc un ordinateur, comme celui que vous utilisez pour travailler, jouer à des jeux vidéo et faire des travaux de programmation. Il existe toutefois une différence majeure : Un serveur utilise normalement toutes ses ressources pour l'application logicielle qu'il exécute. Dans cet exemple, le logiciel est une application web, un programme informatique qui s'exécute sur le serveur.

Dans l'Exemple d'écran affiché pendant qu'une réponse est en préparation, le serveur a reçu une demande de la part d'un navigateur et l'application qui tourne sur le serveur traite l'opération qui en résulte. Cette opération peut consister à interroger une base de données pour obtenir les coordonnées d'un utilisateur dans la banque, ou à communiquer avec un autre serveur pour vérifier une réduction spéciale sur le prochain prêt de l'utilisateur.

Dans cet exemple, nous appelons le navigateur qui tourne sur votre machine l'*application cliente*, ou plus simplement, le *client*. Le client interagit avec le serveur distant.

La communication entre le client et le serveur peut avoir lieu au sein d'un réseau d'entreprise ou à travers le réseau mondial que nous appelons l'internet. Une caractéristique commune de l'interaction client-serveur est qu'un serveur peut établir des multiples relations avec plusieurs clients. Reprenons l'exemple précédent : le site web d'une banque hébergé sur un serveur peut recevoir des milliers de demandes par minute en provenance de plusieurs endroits, chacune d'entre elles émanant d'un utilisateur qui tente d'accéder à son compte bancaire personnel.

Tous les scénarios ne sont pas structurés comme un navigateur interagissant avec un serveur qui effectue la quasi-totalité du traitement. Dans certains cas, le client peut être l'instance principale de traitement ; ce concept est appelé *client lourd*, dans lequel le client stocke et traite la majorité des tâches au lieu de s'appuyer sur les ressources du serveur. Dans notre exemple bancaire, en revanche, le navigateur est un *client léger* qui s'appuie sur le serveur pour calculer et renvoyer des informations via le réseau.

Un exemple de client lourd est l'application de bureau d'un jeu vidéo, où l'essentiel du stockage et du traitement des données se fait localement, en utilisant le GPU, la RAM, le CPU et l'espace disque de l'ordinateur pour traiter les informations. Une telle application dépend rarement d'un serveur externe, surtout si le jeu est joué hors ligne.

Les deux approches présentent des avantages et des inconvénients : Pour un client lourd, l'instabilité du réseau est moins problématique que pour un client léger qui dépend d'un serveur distant, mais les mises à jour logicielles peuvent être plus difficiles à appliquer et le client lourd nécessite plus de ressources informatiques. Pour un client léger, les coûts inférieurs peuvent être un grand avantage. Quel que soit le type de client, la transmission de données personnelles à une application tierce peut poser des problèmes.

Applications web

Une *application web* est un logiciel qui s'exécute sur un serveur, traite les interactions des utilisateurs et est contacté par des clients lourds ou légers via un réseau informatique. Tous les sites web ne sont pas considérés comme des applications web : Les simples pages web statiques sans interactivité ne sont pas considérées comme des applications web car le serveur n'exécute

pas d'application pour traiter les actions demandées par le client.

De nombreuses applications web peuvent être divisées en deux groupes : les *applications monopage* (*single page application* ou SPA) et les *applications multipages* (*multi-page application* ou MPA). Une SPA ne comporte qu'une seule page web, où tous les échanges et chargements de données ont lieu sans qu'il soit nécessaire de rediriger l'utilisateur vers une autre page web à l'intérieur de l'application. Une MPA, contrairement à une SPA, comporte plusieurs pages web. Une modification des données peut soit rafraîchir la page web à l'origine de l'action, soit rediriger l'utilisateur vers une autre page web.

Reprenons l'exemple précédent, dans lequel un utilisateur souhaite consulter les dernières transactions de son compte sur le site web de la banque en ligne. Imaginons qu'une transaction se produise après le chargement de la page web. Si l'application web de la banque est une SPA, la nouvelle transaction sera affichée automatiquement sur la même page web, sans rediriger l'utilisateur vers une nouvelle page. Si l'utilisateur vérifie ses prêts, les nouvelles informations sont également affichées sur la même page, sans qu'il soit nécessaire de rediriger l'utilisateur vers une nouvelle page web. La modification de la page sans redirection de l'utilisateur rend la navigation fluide.

Dans le cas d'une MPA, lorsque l'utilisateur demande la page web du prêt, le serveur doit le rediriger vers un nouvel emplacement, c'est-à-dire une autre page web.

Google Mail (Gmail) est un exemple très connu d'application web SPA. L'application ne redirige pas l'utilisateur vers une page complètement nouvelle lorsque, par exemple, il souhaite afficher le dossier des messages indésirables (*spam*) ; elle se contente de rafraîchir la partie spécifique de l'affichage qui montre tous les messages indésirables, et reste sur la même page web.

D'autre part, une application MPA très connue est Amazon.com, le géant du commerce électronique, où chaque article se trouve sur une page web distincte.

L'un des avantages d'une MPA par rapport à une SPA est qu'il est beaucoup plus facile de recueillir et de mesurer les données analytiques. Ces données sont essentielles pour aider les développeurs à optimiser les résultats de recherche sur internet.

En général, une application web est divisée en deux parties distinctes : la frontale (*frontend*) et la dorsale (*backend*). La partie frontale est la couche de visualisation, où l'utilisateur interagit avec un navigateur en utilisant les éléments de la page en cliquant, en sélectionnant ou en tapant. C'est ici que les données du serveur sont reçues, mises en forme et affichées à l'utilisateur par l'intermédiaire du navigateur.

La partie dorsale est généralement la plus importante d'une application web. Elle comprend la logique commerciale, les gestionnaires de communication, la majorité du traitement et le stockage

des données. Le stockage des données utilise un système de gestion de base de données distinct connecté à la partie dorsale.

La partie frontale et la partie dorsale communiquent entre elles. Les demandes de données sont transmises par la partie frontale à la partie dorsale, et les données renvoyées par la partie dorsale sont reçues, formatées et affichées par la partie frontale à destination de l'utilisateur.

Dans notre exemple simple de recherche de la dernière transaction sur le compte bancaire d'un utilisateur, l'action est interprétée par la partie frontale de l'application, qui s'exécute dans le navigateur sur le bureau de l'utilisateur. Cette demande est ensuite envoyée via l'internet à la partie dorsale de l'application, qui vérifie si l'utilisateur est autorisé à effectuer l'action, récupère les données et renvoie la liste des transactions à la partie frontale chargée dans le navigateur. Le navigateur met alors en forme et affiche les données à l'utilisateur.

Interface de programmation d'application (API)

Aucun logiciel n'est utile s'il ne communique pas avec des composants internes et externes. Comment le client peut-il communiquer avec l'application web ? Comment la partie frontale peut-elle envoyer des données à la partie dorsale ?

Les applications logicielles communiquent entre elles par l'intermédiaire d'une *interface de programmation d'application* (*application programming interface* ou API), fonctionnant sur des *protocoles* de communication Internet basiques. Les protocoles sont des normes et des règles élaborées pour garantir que deux ou plusieurs applications échangent des commandes et des données.

Le principal avantage d'une API est de dissocier les différentes parties de l'application tout en leur permettant de coopérer dans le traitement des données. Les API centralisent également le flux de données dans des canaux prédéfinis, agissant comme une passerelle qui garantit que tout le monde utilise la même voie pour aller et venir. Dans les applications web, les API sont essentielles au fonctionnement de l'application, car elles permettent l'interaction avec l'utilisateur, la fourniture d'informations traitées, les demandes de stockage de données et bien d'autres tâches. Une API peut être utilisée par le client pour demander des actions qui seront exécutées sur le serveur, par exemple.

Revenons à notre exemple d'application web bancaire. Pour se connecter à un compte via une application web, l'utilisateur saisit généralement des données telles que son nom d'utilisateur et son mot de passe dans certains champs de texte et clique sur un bouton "Connexion". Le navigateur saisit ces informations et appelle une API dorsale. L'application web exécutée sur le serveur distant reçoit les données de l'utilisateur, valide l'utilisateur, vérifie le droit de l'utilisateur à obtenir l'accès, et enfin renvoie une réponse au navigateur. Pour que l'utilisateur

puisse communiquer avec le serveur, le client et le serveur doivent obligatoirement envoyer des données dans les deux sens. C'est ce que permettent les API.

Notez que l'application web de la banque n'expose pas d'autres informations sensibles ; elle ne montre à l'utilisateur que les champs autorisés et nécessaires à l'interaction souhaitée. Le reste est caché à l'utilisateur.

La communication entre les API peut être basée sur des conceptions et des protocoles très différents. Le *hypertext transfer protocol* (HTTP) est de loin le protocole le plus fréquemment utilisé dans les applications web. L'*hypertexte* est un texte comportant des liens vers d'autres textes, concept qui constitue la base des liens dans les pages web HTML. Les hyperliens constituent donc la base de la construction des pages web et de leur affichage dans les navigateurs.

Le protocole HTTP a été conçu pour les applications client-serveur, dans lesquelles les ressources sont demandées à un serveur et renvoyées au client via le réseau en utilisant une structure prédéfinie spécifiée par le protocole HTTP.

Dans le cas d'une application web structurée, les ingénieurs logiciels conçoivent l'application avec des parties ou des modules distincts. Cette séparation des responsabilités permet de définir clairement les tâches et les responsabilités, ce qui accélère le développement et améliore la maintenance.

Prenons l'exemple d'une application comportant deux modules internes : l'un qui met en œuvre la logique commerciale et l'autre qui s'appuie sur l'intégration d'un tiers. Cette tierce partie est une entreprise externe qui fournit son API dans un but spécifique, par exemple pour les prévisions météorologiques. Si le serveur météorologique est en panne, il est impossible d'obtenir des détails météorologiques, et si ces données sont cruciales pour le résultat final du traitement, l'utilisateur peut éprouver quelques maux de tête temporaires s'il n'y a pas de source alternative pour les données.

Imaginons maintenant que ce fournisseur tiers soit remplacé et que le nouveau ait une façon différente de gérer la même API. La séparation des modules signifie que les développeurs d'un seul module doivent le mettre à jour. La logique commerciale de l'autre module n'a pas besoin d'être modifiée, ou du moins ne nécessite que des mises à jour minimales.

La nécessité de structures de processus claires influence également la conception des API afin de faciliter leur utilisation. Le concept de *representational state transfer* (REST) est un style architectural avec un ensemble de lignes directrices pour la conception et la mise en œuvre de l'accès aux données dans une application.

Il existe six principes REST. Par souci de simplicité, nous allons en expliquer trois qui sont les plus

pertinents pour cette leçon :

Séparation client-serveur

Le client ne doit connaître que l'URI de la ressource, par lequel la communication avec le serveur s'effectue. Ce principe permet une plus grande flexibilité. Par exemple, lorsque le côté *backend* de l'application est remanié ou qu'une base de données *backend* fait l'objet d'un changement architectural majeur, le *frontend* n'a pas besoin d'être mis à jour en même temps. Il continue simplement à envoyer les mêmes requêtes HTTP au *backend*.

Sans état

chaque nouvelle demande est indépendante des précédentes. Ce n'est pas une coïncidence si le protocole HTTP est largement utilisé pour les applications qui suivent les principes REST, puisque HTTP n'a aucune connaissance des requêtes HTTP précédentes ; pour chaque nouvelle requête, toutes les informations nécessaires doivent être envoyées afin de traiter correctement la requête. Par exemple, une application web qui met en œuvre ce principe ne sait pas si le client est connecté (authentifié). Pour chaque requête HTTP, le client doit donc envoyer un jeton d'authentification. Le serveur peut utiliser ce jeton pour vérifier si la requête doit être bloquée ou traitée.

L'un des principaux avantages de ce principe est qu'il facilite la mise à l'échelle, car le serveur peut traiter des millions de demandes sans vérifier les détails de l'utilisateur.

Architecture en couches

L'application est composée de plusieurs couches, et chaque couche peut avoir sa propre logique et son propre objectif, comme la sécurité ou l'acquisition de données. Le client peut ne jamais savoir combien de couches existent, ou s'il communique directement avec une couche spécifique à l'intérieur de l'application.

Les API qui suivent les principes REST sont appelées *RESTful*, et dans le web moderne, la conception REST est suivie par de nombreuses applications web. Bien qu'une API RESTful ne doive pas nécessairement mettre en œuvre ces principes à l'aide du protocole HTTP, celui-ci est presque universellement utilisé dans le modèle REST en raison de sa robustesse, de sa simplicité et de son omniprésence dans l'environnement du web mondial.

Types d'architectures

Il existe des dizaines de styles et de normes architecturales qui tentent d'organiser les structures des applications web. Comme presque tout dans le monde de l'informatique, il n'y a pas de "gagnant", seulement un ensemble d'avantages et d'inconvénients pour chaque modèle. Un paradigme important est l'*architecture microservice*, qui a été créée comme une alternative à

l'ancienne *architecture monolithique*.

L'architecture microservice est un modèle logiciel composé de multiples *services* interdépendants qui forment ensemble l'application finale. Cette architecture vise à décentraliser le code base : Les multiples couches de logiciels sont divisées en plusieurs applications plus petites pour une meilleure maintenance ([Architecture microservice](#)).

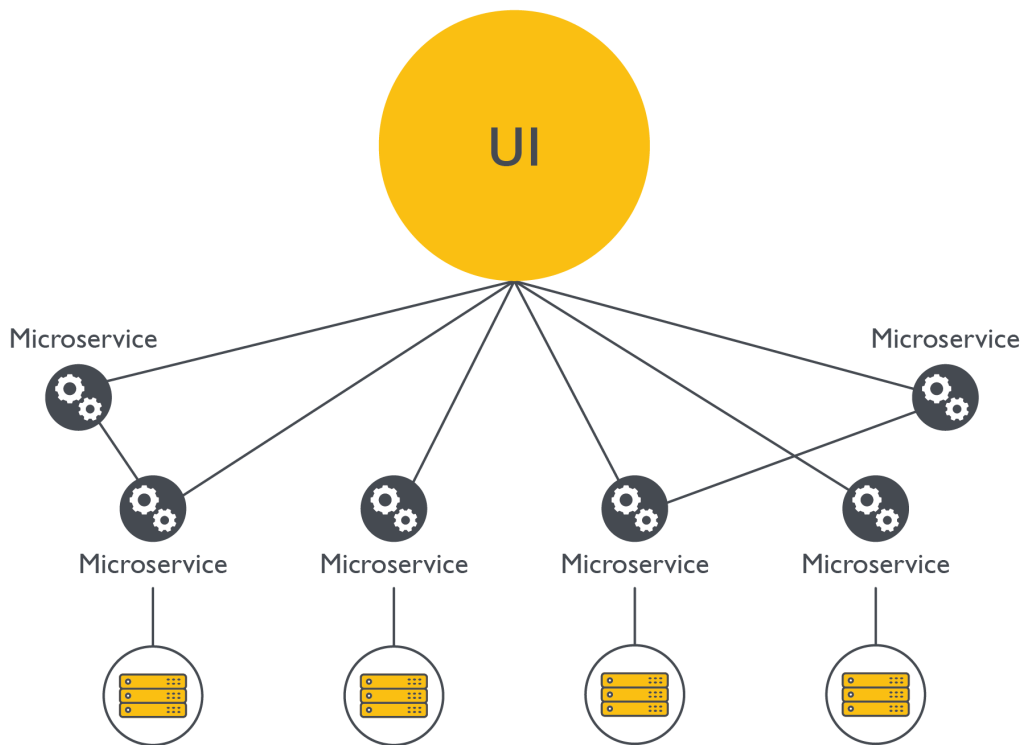


Figure 2. Architecture microservice

En revanche, une architecture monolithique contient tous les services et ressources de l'application dans une seule application ([Architecture monolithique](#)).

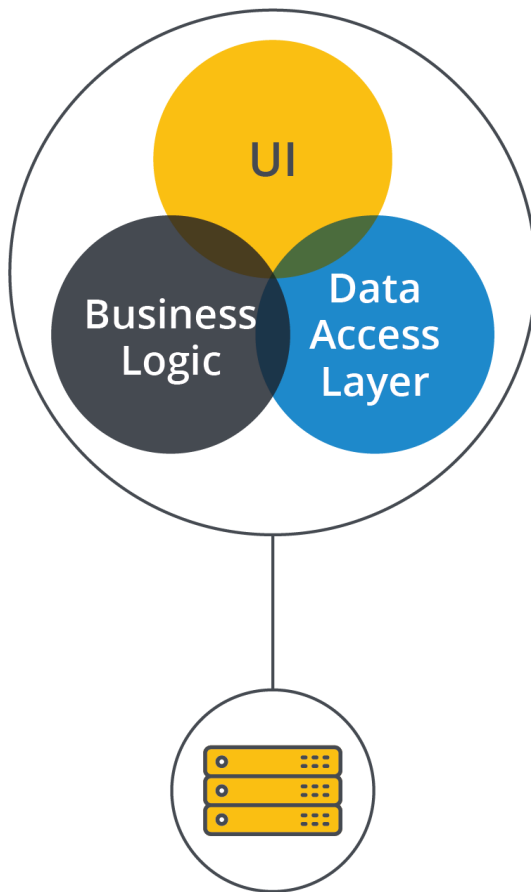


Figure 3. Architecture monolithique

Les images montrent que le modèle microservice est décentralisé et que l'application repose sur plusieurs services, chacun ayant sa propre base de données, son propre code base et même ses propres ressources de serveur. Comme son nom l'indique, chaque microservice doit être plus petit que son homologue monolithique, qui prend la responsabilité de tous les services.

L'application monolithique encapsule toutes ses ressources dans une seule unité. Toute la logique commerciale, les données et le code base sont centralisés dans un énorme bloc, c'est pourquoi on l'appelle un "monolithe".

Les utilisateurs remarquent à peine si une application web fonctionne selon un modèle monolithique ou microservice ; le choix devrait être transparent. Notre application bancaire, par exemple, pourrait être un monolithe dans lequel toute la logique commerciale concernant les paiements, les transactions, les prêts, etc. se trouve dans le même code base qui s'exécute sur un ou plusieurs serveurs. En revanche, si l'application bancaire utilise un style de microservice, elle dispose probablement d'un microservice dédié au traitement des paiements et d'un autre

microservice réservé à l'octroi de prêts. Ce dernier microservice appelle encore un autre microservice pour analyser la probabilité que le demandeur n'effectue pas le paiement. L'application pourrait comporter des milliers de petits services.

L'approche monolithique nécessite davantage de frais généraux de maintenance lorsque l'application prend de l'ampleur, en particulier lorsque plusieurs équipes codent dans le même code base. Compte tenu de la centralisation des ressources logicielles, il est très probable qu'une équipe modifie quelque chose qui casse la partie de l'application appartenant à une autre équipe. Cela peut constituer un véritable casse-tête pour les grandes équipes, particulièrement lorsqu'elles sont très nombreuses.

Les microservices sont beaucoup plus flexibles à cet égard, car chaque service est géré par une seule équipe. Une équipe peut, bien entendu, gérer plus d'un service. Les changements de code sont faciles à effectuer et la concurrence entre les ressources n'est pas un réel problème. Comme chaque service est interconnecté, tout point de défaillance peut avoir un impact négatif sur l'ensemble de l'application. En outre, étant donné que de multiples instances de bases de données, serveurs et API externes communiquent entre eux, la résilience de l'ensemble de l'application n'est que le reflet de son microservice le plus faible.

L'un des avantages de l'approche monolithique est de disposer d'une source de données centralisée, ce qui permet d'éviter plus facilement la duplication des données. Cette approche permet également de réduire la consommation des ressources cloud, car un grand serveur nécessite moins de ressources informatiques que plusieurs serveurs décentralisés. Une application de microservices de taille à peu près équivalente fait peser une charge plus importante sur le cloud.

Exercices guidés

1. Quelles sont les principales différences entre un client lourd et un client léger ?

2. Est-il correct de supposer que tout site web est une application web ?

3. Qu'est-ce que le modèle REST ?

4. Quel est le modèle privilégié pour le développement d'applications web modernes et de grande envergure avec plusieurs équipes de développement ? Pourquoi ?

5. Quel est le protocole le plus couramment utilisé pour échanger des données entre applications web ?

6. Citez deux inconvénients des applications multipages par rapport aux applications monopage.

7. Décrivez un avantage d'un système monolithique par rapport à un système de microservices, et un avantage du système de microservices par rapport à un système monolithique.

Exercices d'exploration

1. En 2021, le Rover Persévérance de la NASA s'est posé sur Mars, l'un de ses objectifs étant de déterminer si la vie a déjà existé sur cette planète. Bien que le Rover puisse être contrôlé à distance ici sur Terre, il peut également se contrôler lui-même dans la plupart des situations. Pourquoi est-ce une bonne idée de projeter un tel Rover en tant que gros client ?

2. Prenons l'exemple d'une voiture personnelle moderne et autonome qui se connecte à un serveur externe pour échanger des données. Doit-il s'agir d'un client lourd ou d'un client léger ?

Résumé

Cette leçon explique les concepts fondamentaux de l'architecture logicielle pour les applications web. Elle explique comment elles sont généralement structurées et organisées, et les principales différences entre le modèle monolithique et le modèle microservice. Nous avons abordé les concepts de serveurs et de clients, ainsi que les bases de la communication des applications web entre les clients et les autres programmes logiciels.

Réponses aux exercices guidés

1. Quelles sont les principales différences entre un client lourd et un client léger ?

Un client lourd n'a pas besoin d'une connexion constante à un serveur distant qui renvoie des informations critiques au client en cours d'exécution. Le client léger dépend fortement des informations traitées par une source externe. Une autre différence réside dans le fait qu'un client lourd est responsable de la majeure partie du traitement des données, ce qui nécessite davantage de ressources informatiques que son homologue léger.

2. Est-il correct de supposer que tout site web est une application web ?

Non. Il existe des sites web qui ne sont pas des applications logicielles. Une application web interagit avec l'utilisateur, qui peut saisir des données et utiliser des fonctionnalités web en temps réel. Les sites web simples, tels qu'une publicité pour un événement social, qui fonctionne comme une bannière web, ne sont pas des applications web. Ces sites web non interactifs sont plus faciles à entretenir et nécessitent de faibles ressources informatiques pour héberger et diffuser les pages web. Une application web nécessite des ressources informatiques beaucoup plus importantes, des serveurs plus robustes et des fonctionnalités qui gèrent les utilisateurs, tels qu'un accès restreint et un stockage permanent des données.

3. Qu'est-ce que le modèle REST ?

Le modèle REST est un modèle d'architecture logicielle qui fournit aux applications un guide de développement permettant d'améliorer la convivialité, la clarté et la maintenabilité. L'un des principes énoncés dans l'ensemble des lignes directrices REST est l'*architecture en couches*, utilisée principalement à des fins de cohésion et pour réduire la dépendance des différents composants internes des API.

4. Quel est le modèle privilégié pour le développement d'applications web modernes et de grande envergure avec plusieurs équipes de développement ? Pourquoi ?

Le modèle logiciel des microservices fournit un cadre flexible dans lequel les équipes collaborent sur la même application logicielle, ce qui facilite la concurrence pour deux équipes ou plus qui maintiennent une grande application web. Le cadre étant décentralisé, chaque équipe peut mettre à jour un domaine d'activité spécifique sans avoir à mettre à jour d'autres composants.

5. Quel est le protocole le plus couramment utilisé pour échanger des données entre applications web ?

Le protocole HTTP est le plus utilisé pour l'échange de données et de commandes entre les serveurs et les clients.

6. Citez deux inconvénients des applications multipages par rapport aux applications monopage.

Une application multipage recharge tous les éléments de la page web lorsque l'utilisateur déclenche certaines actions, au lieu de ne mettre à jour que les éléments modifiés. Les performances souffrent de cette conception. Un autre inconvénient d'une MPA est la lourdeur de l'interactivité avec l'utilisateur, chaque chargement de page entraînant une perte de convivialité. En revanche, l'effet visuel peut être continu dans une SPA.

7. Décrivez un avantage d'un système monolithique par rapport à un système de microservices, et un avantage du système de microservices par rapport à un système monolithique.

Un système monolithique peut faciliter l'administration des données, car celles-ci se trouvent dans une seule grande base de données, au lieu d'être dispersées dans plusieurs bases de données. Une application microservice, en revanche, peut améliorer le développement et la maintenance du code ; plusieurs équipes peuvent travailler sur différentes logiques d'entreprise sans bloquer les progrès des autres équipes.

Réponses aux exercices d'exploration

1. En 2021, le Rover Persévérance de la NASA s'est posé sur Mars, l'un de ses objectifs étant de déterminer si la vie a déjà existé sur cette planète. Bien que le Rover puisse être contrôlé à distance ici sur Terre, il peut également se contrôler lui-même dans la plupart des situations. Pourquoi est-ce une bonne idée de projeter un tel Rover en tant que gros client ?

Le temps nécessaire pour qu'un signal de communication soit envoyé de la Terre et reçu sur Mars peut varier en fonction de la position de ces planètes, mais il peut prendre jusqu'à vingt minutes. Il est donc impossible de commander et de contrôler un Rover distant en mouvement, surtout si l'on tient compte des situations inattendues. Idéalement, le Rover devrait se commander lui-même dans la plupart des situations. Pour ce faire, on utilise l'apprentissage de l'intelligence artificielle (IA) (apprentissage automatique), de sorte que le Rover soit plus indépendant des commandes manuelles. Pour que cela soit possible et que le Rover dépende moins des signaux distants, il a été prévu que le Rover dispose de ses propres ressources et que la majorité des processus informatiques soient exécutés localement, ce qui correspond à la définition d'un client lourd.

2. Prenons l'exemple d'une voiture personnelle moderne et autonome qui se connecte à un serveur externe pour échanger des données. Doit-il s'agir d'un client lourd ou d'un client léger ?

Un véhicule autonome pourrait déléguer le traitement des données lourdes à un serveur externe et fiable, mais cela risquerait d'entraîner des périodes hors ligne lorsque le traitement des données critiques est nécessaire. Il est donc impératif que le véhicule autonome traite la majorité des tâches, ce qui exige qu'il soit un client lourd avec de multiples redondances.



051.3 Informatique sur site et dans le cloud

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 051.3

Valeur

1

Domaines de connaissance les plus importants

- Comprendre les concepts de l'informatique sur site et de l'informatique dans le cloud
- Comprendre les modèles courants d'exploitation de l'informatique dans le cloud
- Comprendre les types courants de services dans le cloud
- Comprendre les principaux avantages et risques de l'informatique dans le cloud et de l'infrastructure informatique sur site

Liste partielle de termes, fichiers et utilitaires utilisés

- Informatique dans le cloud
- Infrastructure informatique sur site
- Centre de données
- Cloud public, privé et hybride
- Infrastructure en tant que service (IaaS), plateforme en tant que service (PaaS), logiciel en tant que service (SaaS)
- Modèles de coûts
- Sécurité
- Propriété des données
- Disponibilité des services



**Linux
Professional
Institute**

Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	051 Fondements des logiciels
Objectif :	051.3 Informatique sur site et dans le cloud
Leçon :	1 sur 1

Introduction

De nos jours, tout le monde parle du cloud. Les entreprises évaluent les services d'une série de fournisseurs — y compris de grandes sociétés comme Amazon.com et Microsoft — pour voir ce que le cloud peut offrir.

Mais l'idée de l'informatique dans le cloud ne remonte qu'à la fin des années 2000. Et le terme est vague — si vague que de nombreuses personnes, y compris la Free Software Foundation, en découragent l'utilisation.

Cependant, l'idée de l'informatique dans le nuage englobe certains concepts utiles spécifiques et constitue une tendance cruciale de l'informatique moderne. Cette leçon examine ce qu'est le cloud et, à un niveau élevé, comment il fonctionne à la fois techniquement et financièrement. Nous examinerons les avantages et les risques du cloud. En cours de route, nous verrons pourquoi ce sujet est lié à l'open source.

Informatique sur site et dans le cloud

Si vous travaillez ou étudiez dans une organisation qui possède plus de deux systèmes

informatiques, il est presque certain qu'elle dispose d'un *centre de données* : une pièce séparée pour contenir les serveurs de l'organisation, généralement fermée à clé et climatisée. Un centre de données *sur site* (ou *on-premise*) est simplement un centre que l'organisation maintient pour son propre usage.

Il existe plusieurs alternatives à l'exploitation d'un centre de données sur site. Pendant des décennies, avant la tendance que nous appelons "informatique dans le cloud" (ou "*_cloud computing_*"), les entreprises mettaient en place des centres de données qui hébergeaient des ordinateurs pour le compte de leurs clients. Ainsi, vous pourriez obtenir une licence pour cinq serveurs auprès de l'entreprise et lui donner diverses spécifications pour le processeur, la mémoire et le stockage, puis télécharger votre logiciel sur les serveurs. Ce modèle d'entreprise est appelé *hébergement à distance* (ou *remote hosting*).

L'hébergement à distance offre de nombreux avantages. L'organisation peut effectivement externaliser l'expertise administrative de l'achat et de l'installation des serveurs à l'entreprise d'hébergement à distance, qui peut obtenir des achats en gros. En d'autres termes, vous évitez la responsabilité d'avoir une infrastructure informatique sur site. L'entreprise d'hébergement à distance assure la sécurité physique, ce qui libère également le client de ce souci. Enfin, l'installation d'un nouveau serveur dans une entreprise d'hébergement à distance est beaucoup plus rapide que l'achat, l'expédition et l'installation du serveur sur site.

Une organisation peut également conserver un centre de données sur site tout en acquérant des licences pour davantage de serveurs dans l'environnement distant afin de se remettre d'un sinistre ou de fournir une puissance de calcul supplémentaire pendant les périodes d'utilisation intensive.

Il est à noter que l'informatique à distance suppose la présence d'un réseau rapide et fiable. Nous examinerons cette question ainsi que d'autres avantages et faiblesses dans une autre section.

Tous les avantages de l'hébergement à distance s'appliquent également à l'informatique dans le cloud. L'informatique dans le cloud est techniquement différente parce qu'aucun ordinateur physique particulier n'est réservé à votre organisation. Au lieu de cela, le fournisseur de l'informatique dans le cloud fait fonctionner plusieurs systèmes pour plusieurs clients sur chaque système physique, en utilisant une couche supplémentaire de logiciel connue sous le nom de *machines virtuelles*.

Ainsi, un service de cloud gère un centre de données comme n'importe quelle autre organisation, mais il sert d'autres organisations à la place (ou en plus) de lui-même. Le centre de données stocke des milliers d'ordinateurs physiques. Sur chaque ordinateur physique, il exécute un système d'exploitation (généralement appelé *hyperviseur*) qui prend en charge plusieurs machines virtuelles. Chaque machine virtuelle peut être créée et supprimée rapidement. Chaque machine

virtuelle prend en charge un système d'exploitation exécuté par un client (Informatique dans le cloud).

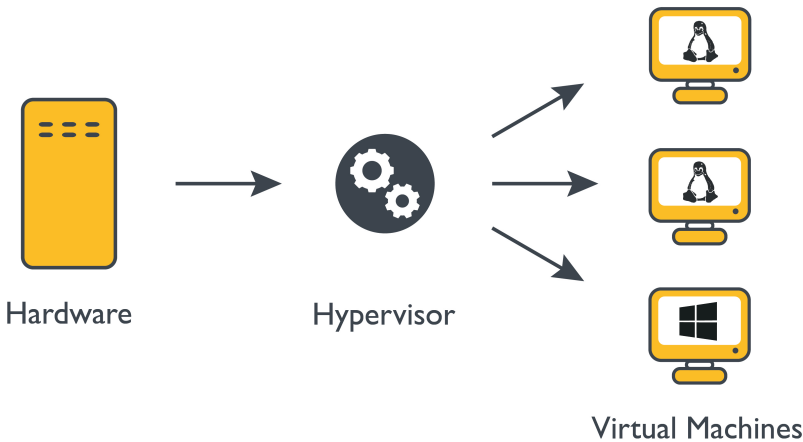


Figure 4. Informatique dans le cloud

Quel est le rôle des logiciels libres et de l'open source ? Lorsqu'une entreprise exploite un grand nombre d'ordinateurs et déploie des systèmes d'exploitation à la volée, il est important de ne pas se perdre dans les problèmes de licences. Bien qu'il existe des modèles de licence pour les systèmes d'exploitation propriétaires dans le cloud, ils sont plus compliqués que la simple utilisation d'une machine virtuelle et d'un système d'exploitation open source. De plus, les logiciels libres ne coûtent généralement rien.

Les débuts de l'informatique dans le cloud remontent généralement au lancement par Amazon.com d'Amazon Web Services (AWS) en 2006. Il existe aujourd'hui des dizaines de sociétés spécialisées dans l'informatique dans le cloud, y compris des offres de grands fournisseurs tels que Microsoft, Google, Alibaba et IBM. AWS reste l'offre la plus importante. Les fournisseurs se livrent une concurrence féroce pour développer de nouvelles fonctionnalités et de nouveaux services, car ils sont tous très attachés aux coûts et à la fiabilité.

Les avantages de l'informatique en nuage s'appuient sur ceux de l'informatique à distance. Les coûts sont moins élevés parce qu'un système physique peut faire fonctionner de nombreux serveurs pour de nombreux clients et peut être occupé en permanence. Un client qui a besoin rapidement de plus de puissance de calcul pour faire face à un pic d'utilisation peut créer de nouveaux systèmes en quelques secondes. Les systèmes peuvent être gérés automatiquement via une interface de programmation d'applications (API). Là encore, nous examinerons de plus près les avantages et les risques.

Modèles communs de fonctionnement du cloud

Avant de détailler les modèles de fonctionnement, il convient de noter que de nombreuses entreprises ont adopté des modèles de cloud qui modifient totalement leur façon de programmer et d'offrir des services. Au lieu de mettre à jour chaque application une ou deux fois par an, les entreprises permettent des mises à jour rapides. Elles peuvent le faire parce que le cloud leur permet d'arrêter les machines virtuelles et d'en démarrer de nouvelles presque instantanément avec la nouvelle version de l'application. L'organisation peut également évoluer rapidement, c'est pourquoi elle aime diviser les applications en plusieurs parties modulaires, parfois appelées *microservices*.

Mais dans cette section, nous nous concentrerons sur les modèles de cloud de tous les jours.

Le modèle de coût du cloud est très différent de celui des coûts sur site. Les centres de données sur site nécessitent l'achat unique d'un serveur, ainsi que des coûts courants pour l'alimentation électrique, la climatisation et l'administration. Ces coûts disparaissent lorsque vous obtenez une licence pour un système auprès d'un fournisseur de services en cloud. Au lieu de cela, vous êtes facturé pour ce que vous utilisez. Les fournisseurs de services en cloud divisent votre utilisation de l'ordinateur en périodes de temps et vous facturent pour chaque période. Ils facturent également la quantité de données que vous stockez sur leurs systèmes.

Jusqu'à présent, nous avons parlé des fournisseurs qui offrent de la puissance de calcul à leurs clients ; c'est ce qu'on appelle le *cloud public*. Mais il peut aussi y avoir un *cloud privé*. Certaines grandes entreprises gèrent leurs propres centres de données sur site comme un cloud. Elles ne fournissent des services qu'à leurs propres départements ou subdivisions, mais elles traitent chacun de leurs départements comme un client d'un fournisseur de cloud. Le centre de données garde la trace de la quantité de temps de calcul, de données, etc. utilisée par chaque département et lui facture cette utilisation.

De nombreuses organisations utilisent plusieurs services en cloud pour diverses raisons, par exemple pour se protéger contre les défaillances d'un fournisseur, conserver des données dans une certaine région géographique ou profiter des fonctions spéciales offertes par un fournisseur particulier. En outre, il est courant de maintenir à la fois un centre de données sur site et des serveurs dans le cloud, une pratique appelée *informatique hybride* (ou *hybrid computing*).

Un client qui s'inscrit à un service de cloud peut choisir les régions géographiques dans lesquelles il souhaite travailler. Par exemple, Amazon propose actuellement des régions à l'ouest et à l'est des États-Unis, plusieurs régions en Europe et d'autres régions dans toutes les parties du monde. Normalement, vous devriez choisir la région la plus proche de chez vous. Cependant, de nombreuses organisations souhaitent opérer dans plusieurs régions parce qu'elles ont une portée internationale. Les organisations doivent parfois conserver des données dans un endroit

particulier pour se conformer au règlement général sur la protection des données (RGPD) en Europe ou à la loi sur la protection des informations personnelles (*Personal Information Protection Law PIPL*) en Chine.

Chaque région est normalement subdivisée en *zones* ou *zones de disponibilité*. Il est recommandé de gérer plusieurs zones au cas où un désastre entraînerait l'arrêt d'une zone.

Bien que le cloud soit réputé pour le partage d'ordinateurs physiques entre plusieurs clients, certains fournisseurs peuvent consacrer un seul ordinateur à un client particulier qui s'inquiète de la sécurité. Comme aucune autre organisation n'utilise l'ordinateur, le client se sent un peu plus en sécurité lorsqu'il exécute ses services sensibles et télécharge ses données dans le cloud. Cette option rapproche l'informatique dans le cloud de l'hébergement à distance à l'ancienne.

Types communs de services dans le cloud

À différents niveaux, le cloud se présente de manière très différente et s'adresse à différents types d'utilisateurs.

Infrastructure en tant que service (Infrastructure as a Service - IaaS) est la catégorie dont s'occupent généralement les administrateurs de systèmes. L'IaaS fournit uniquement le matériel et les logiciels qui prennent en charge les machines virtuelles. Il appartient aux administrateurs système du client de charger un système d'exploitation et les applications souhaitées sur la machine virtuelle. L'administrateur système gère presque tout de la même manière que dans un centre de données sur site.

Plateforme en tant que service (Platform as a Service - PaaS) est une invention plus récente, utilisée principalement par les programmeurs. Ici, le programmeur ne se préoccupe pas du système d'exploitation et ne doit pas charger les bibliothèques que le programme utilise. Tout cela est fourni par le fournisseur de services dans le cloud. Le programmeur se contente de télécharger des fonctions qui s'exécutent sur la plateforme. L'informatique *sans serveur (serverless)* est un concept connexe.

Logiciel en tant que service (Software as a Service - SaaS) est une application fonctionnant sur un système dans le cloud. Chaque fois que vous vous connectez à un site d'un réseau social, que vous commandez un article dans un magasin en ligne, que vous visitez une page web pour entrer vos heures dans un système de suivi des emplois ou que vous remplissez un formulaire sur un site gouvernemental, vous utilisez un SaaS. La majeure partie de l'application s'exécute sur le système distant, et la seule partie de l'application qui s'exécute sur votre ordinateur est la page web affichée par votre navigateur.

Base de données en tant que service (Database as a Service - DaaS) est souvent ajoutée aux

catégories précédentes. Un service de données, le S3 d'Amazon, a été en fait la première offre en cloud. Une offre DaaS peut simplement consister en une instance d'un serveur de base de données populaire tel que MySQL, Oracle ou MongoDB fonctionnant dans le cloud. Les grands fournisseurs de services dans le cloud proposent également des bases de données propriétaires qui s'exécutent uniquement dans leurs offres de services dans le cloud. Dans tous les cas, vous lisez et écrivez la base de données comme si vous l'aviez sur vos systèmes.

D'autres variantes de ces catégories de base sont proposées par certaines entreprises, comme la sécurité en tant que service (*Security as a Service*).

Principaux avantages et risques du cloud et de l'infrastructure informatique sur site

Avant d'examiner en détail le cloud, essayons une analogie. Exploiter un centre de données sur site, c'est comme acheter une maison. Si le sous-sol est inondé ou si la chaudière ne fonctionne plus, vous devez trouver quelqu'un pour la réparer. En revanche, l'hébergement à distance et le cloud sont comparables à la location d'un appartement : Le propriétaire est responsable de la réparation de la chaudière. En outre, dans le cloud, vous pouvez rapidement ajouter et supprimer des charges de données, tout comme vous pouvez changer d'appartement plus rapidement que vous ne pouvez changer la maison que vous possédez.

Dans un appartement, le propriétaire peut même fournir des appareils et des meubles. Dans notre analogie, il s'agit des nombreux services offerts par les fournisseurs de services dans le cloud, tels que les bases de données et les outils analytiques.

Nous pouvons maintenant examiner les avantages et les risques liés à l'utilisation du cloud, à la place ou en complément de votre propre centre de données.

Flexibilité c'est probablement la raison la plus convaincante de passer au cloud. Si vous êtes un détaillant qui a besoin d'exploiter davantage de serveurs à l'approche de Noël, ou un comptable spécialisé dans la préparation des déclarations d'impôts qui réalise l'essentiel de son activité pendant la saison des impôts, vous voudrez utiliser le cloud pour mettre en service de nouveaux serveurs en un clin d'œil, puis supprimer leurs machines virtuelles par la suite.

Coûts ils peuvent être moins élevés dans le cloud pour plusieurs raisons. Vous partagez un serveur physique avec de nombreuses autres applications, de sorte que les ordinateurs sont utilisés plus efficacement. Les fournisseurs de services dans le cloud étant de grande taille, ils peuvent réaliser des économies d'échelle au niveau des achats, de l'administration, du refroidissement et d'autres exigences en matière d'infrastructure. Enfin, les clients sont libérés de nombreuses tâches administratives — même si l'administration des systèmes ne disparaît pas pour autant. Les clients ont toujours besoin d'administrateurs système pour créer et télécharger

leurs logiciels (appelés *instances* dans le cloud), pour autoriser les utilisateurs et pour d'autres tâches liées aux opérations commerciales. Les administrateurs système doivent apprendre l'API du fournisseur et les règles d'utilisation du service, ce qui représente un coût de formation que vous devez prendre en compte dans vos plans.

D'un autre côté, il faut faire attention à l'utilisation que l'on fait du cloud. Il peut être difficile de suivre la quantité d'énergie consommée par les machines lorsque l'on peut rapidement faire tourner les serveurs, surtout si l'on automatise la mise à l'échelle. Vous risquez de vous retrouver avec une facture désagréablement élevée à la fin de la période.

Le cloud est-il plus efficace en termes d'émissions de carbone que l'utilisation de nos propres ordinateurs ? Des études montrent que les fournisseurs de services dans le cloud peuvent faire fonctionner leurs systèmes de manière beaucoup plus efficace que vous et moi. Mais nous devons communiquer avec ces systèmes par l'intermédiaire d'un réseau, ce qui nécessite beaucoup d'électricité pour alimenter tous les équipements de mise en réseau. Malheureusement, le cloud augmente donc notre empreinte carbone.

Disponibilité des services elle est parfois meilleure avec le cloud. Certes, si vous dépendez de votre propre centre de données sur site, vous êtes vulnérable à toutes sortes de problèmes allant des catastrophes naturelles aux saboteurs internes malveillants. Mais les centres de données dans le cloud ne sont pas à l'abri d'une panne. Vous devez donc tirer parti des différentes zones de disponibilité et répartir les risques. Il existe des outils qui vous permettent de basculer vos services d'une zone défaillante à une zone fonctionnelle.

Si vous utilisez les services offerts par le fournisseur de services dans le cloud, comme une base de données dans le cloud, vous êtes vulnérable aux bogues de ce service. Bien entendu, vous pouvez également être victime de bogues dans le logiciel que vous chargez dans votre système.

Le risque le plus important lié à l'utilisation des services des fournisseurs est le verrouillage. Vous pouvez généralement trouver un outil de conversion automatisé pour transférer vos données du système du fournisseur vers un nouveau système, mais l'outil peut ne pas faire un travail complet.

Sécurité elle peut être meilleure dans le cloud, car le personnel du fournisseur est probablement plus expert que votre personnel de sécurité. D'un autre côté, les fournisseurs de services dans le cloud sont importants et bien connus, ce qui constitue des cibles évidentes pour les attaques. En outre, l'ajout d'un logiciel supplémentaire—l'hyperviseur qui contrôle les machines virtuelles—introduit un nouveau risque potentiel. Des chercheurs ont découvert des vulnérabilités dans les hyperviseurs.

Bien que le client reste le propriétaire légal de ses données, le stockage des données dans le cloud les rend théoriquement plus vulnérables. En général, le client chiffre les données pour les protéger en cas d'intrusion. Les réglementations en matière de protection de la vie privée, telles

que le RGPD mentionné précédemment, exigent que les données soient stockées dans un centre de données situé dans une région considérée comme sûre.

En fin de compte, la plupart des attaques de sécurité commencent à un niveau élevé, comme l'envoi d'un courriel contenant un logiciel malveillant à un employé qui ne se doute de rien. Cela n'a pas d'importance, que vous fonctionniez sur site ou dans le cloud. Mais un intrus malveillant qui s'empare du compte d'un employé n'ira pas beaucoup plus loin s'il ne peut pas tirer parti des vulnérabilités de vos serveurs ; là encore, il n'est pas certain que le fait de fonctionner dans le cloud fasse une grande différence, car la plupart des vulnérabilités se trouvent dans le logiciel plutôt que dans le service dans le cloud.

Enfin, tenez compte des coûts liés à la bande passante et au réseau. Vos clients, et probablement votre personnel, communiquent avec des serveurs qui peuvent se trouver à des centaines de kilomètres. Si le réseau de connexion n'est pas fiable ou est lent, les performances des serveurs situés dans le cloud seront inférieures à celles de votre centre de données sur site. De nos jours, tout le monde se connecte avec des employés à distance, à des services SaaS et à d'autres systèmes géographiquement éloignés. Les performances de votre réseau affecteront presque tout ce que vous faites, que vous soyez ou non dans le cloud.

Exercices guidés

1. Pourquoi un ordinateur physique dans un centre cloud est-il utilisé plus efficacement qu'un ordinateur dans un centre de données traditionnel sur site ?

2. Qu'est-ce qu'un cloud hybride ?

3. Quel type d'informatique dans le cloud nécessite le plus souvent un travail de la part d'un administrateur système du côté du client ?

4. Comment protéger votre service contre les pannes si vous faites appel à un fournisseur de services dans le cloud ?

Exercices d'exploration

1. Comparez les différents types de coûts liés à l'exploitation de vos serveurs dans un cloud à ceux liés à l'exploitation de vos serveurs sur site.

2. Vous travaillez au Moyen-Orient, mais vous avez de nombreux clients en Europe et en Extrême-Orient. Décrivez où vous placeriez vos services dans une offre cloud.

Résumé

Cette leçon a décrit le fonctionnement de l'informatique dans le cloud et les compromis entre l'utilisation du cloud et l'exploitation de systèmes dans votre propre centre de données, sur site. Vous avez appris les différents modèles d'entreprise et de coûts, y compris les différences entre les clouds publics, privés et hybrides. Vous avez également appris les différents types d'offres de cloud et ce à quoi chacune d'entre elles sert.

Réponses aux exercices guidés

1. Pourquoi un ordinateur physique dans un centre cloud est-il utilisé plus efficacement qu'un ordinateur dans un centre de données traditionnel sur site ?

Dans le cloud, chaque ordinateur peut exécuter plusieurs instances de systèmes d'exploitation, et même des instances téléchargées par différents clients. L'ordinateur est donc plus souvent utilisé.

2. Qu'est-ce qu'un cloud hybride ?

Un cloud hybride utilise à la fois les centres de données d'un fournisseur de cloud et un ou plusieurs centres de données sur site.

3. Quel type d'informatique dans le cloud nécessite le plus souvent un travail de la part d'un administrateur système du côté du client ?

L'infrastructure en tant que service (IaaS) exige du client qu'il effectue l'administration du système pour des tâches telles que la création et le téléchargement d'instances du système d'exploitation et d'applications.

4. Comment protéger votre service contre les pannes si vous faites appel à un fournisseur de services dans le cloud ?

Choisissez plusieurs zones dans chaque région où vous gérez votre service, car il est très peu probable que plusieurs zones tombent en panne simultanément.

Réponses aux exercices d'exploration

1. Comparez les différents types de coûts liés à l'exploitation de vos serveurs dans un cloud à ceux liés à l'exploitation de vos serveurs sur site.

Dans un cloud, vous payez pour l'utilisation du processeur et le stockage des données pour chaque période mesurée par le fournisseur. Mais vous ne payez pas de frais de matériel. Sur place, vous devez payer le coût fixe du matériel, ainsi que d'autres équipements tels que la climatisation, et les coûts récurrents tels que l'électricité et la maintenance physique.

2. Vous travaillez au Moyen-Orient, mais vous avez de nombreux clients en Europe et en Extrême-Orient. Décrivez où vous placeriez vos services dans une offre cloud.

Utilisez une région du Moyen-Orient pour vos propres bureaux et vos clients du Moyen-Orient. Une région en Europe est importante pour se conformer au règlement général sur la protection des données (RGPD). Vous pourriez avoir besoin d'une région en Chine pour vous conformer à la loi chinoise sur la protection des informations personnelles (PIPL). Dans tous les cas, le fait d'avoir des régions en Extrême-Orient et en Europe est précieux pour améliorer les performances lors des interactions avec les clients dans ces pays.

Dans chaque région, choisissez plusieurs zones pour vous protéger contre la défaillance d'une seule zone.



Thème 052: Licences de logiciels à code source ouvert



052.1 Concepts des licences de logiciels à code source ouvert

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 052.1

Valeur

3

Domaines de connaissance les plus importants

- Comprendre les définitions des logiciels à code source ouvert et des logiciels libres
- Connaître les autres types de logiciels gratuits
- Connaître les événements importants de l'histoire de l'open source
- Comprendre ce qu'est une licence et quels sont les droits généralement gérés par les licences
- Comprendre comment les logiciels existants peuvent être utilisés pour créer des travaux dérivés
- Comprendre les compatibilités et incompatibilités des licences
- Comprendre les licences doubles et les licences conditionnelles
- Comprendre les conséquences des violations de licence
- Comprendre les principes du droit d'auteur et du droit des brevets et la manière dont ils sont affectés par les licences de logiciels à code source ouvert

Liste partielle de termes, fichiers et utilitaires utilisés

- Définition des logiciels libres selon la Free Software Foundation (FSF)
- Définition des logiciels à code source ouvert selon l'Open Source Initiative (OSI)
- Licences
- Contrats

- Logiciels du domaine public
- Logiciel gratuit (Freeware)
- Logiciel à contribution (Shareware)
- Régisseurs de licence
- Autorisations d'utiliser, de modifier et de distribuer le code et les logiciels
- Travaux dérivés et réutilisation du code
- Logiciels à code source fermé / propriétaires
- Distribution payante
- Distribution de logiciels modifiés et non modifiés
- Hébergement de logiciels en tant que service payant
- Compatibilité des licences
- Licences doubles et multiples
- Licences conditionnelles
- Brevets de logiciels
- Octroi de licences de brevets explicites et implicites



Linux
Professional
Institute

Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	052 Licences de logiciels à code source ouvert
Objectif :	052.1 Concepts des licences de logiciels à code source ouvert
Leçon :	1 sur 1

Introduction

Tout développeur de logiciel apprécie qu'un problème soit résolu efficacement auparavant. Si la solution est disponible en ligne, le réflexe compréhensible est de l'essayer : copier ou lier le code source existant dans son propre code, le tester et le laisser là s'il fonctionne — puis l'oublier.

Mais une certaine prudence s'impose. Le plus souvent, le code source des logiciels disponibles sur internet est couvert par une *licence* de logiciel libre et code source ouvert (FOSS *Free and Open Source Software*). Ce chapitre aborde certains concepts de base des logiciels FOSS d'un point de vue juridique et explique pourquoi il est préférable de ne pas considérer le code source des logiciels FOSS comme acquis, mais plutôt d'observer attentivement les conditions des licences.

Pour comprendre vos obligations légales en matière de logiciels, il faut savoir que presque tous les logiciels sont régis par une licence. Une licence est une sorte de contrat. Tous les logiciels, y compris les sites web, doivent être distribués avec une version écrite de leur licence (souvent appelée "termes et conditions"). Certains programmes et sites web vous obligent à cocher une case indiquant que vous avez lu leurs termes et conditions (et vous devriez le faire, bien que la plupart des gens ne le fassent jamais). Dans tous les cas, vous acceptez implicitement la licence en utilisant

le logiciel.

Définitions des logiciels open source et des logiciels libres

Les logiciels libres et à code source ouvert existent depuis un certain temps, et les deux termes sont souvent associés. Cependant, il existe des divergences entre ce que certains déclarent être des “logiciels libres” et la signification formelle des *logiciels libres*, ainsi que des *logiciels à code source ouvert*. Spoiler : Le terme “logiciel à code source ouvert” ne signifie pas simplement que n’importe qui peut consulter le code source — il s’agirait alors d’un “logiciel à code source disponible”. Le logiciel à code source ouvert est bien plus que cela.

Les logiciels libres et à code source ouvert s’opposent à d’autres types de logiciels considérés comme *propriétaires* ou à *code source fermé* parce qu’ils n’offrent pas toutes les libertés dont il est question dans cette leçon.

Probablement, la définition la plus souvent citée du logiciel libre est la suivante :

Think of “free speech”, not “free beer.”

— Richard Stallman, *Selling Free Software*

que nous pouvons traduire par “Pensez à la *liberté d’expression* et non pas à la *bière gratuite*”. Dans les sections suivantes, nous analyserons les éléments fertilisants de cette déclaration.

Libre comme la parole : La vraie liberté pour les utilisateurs

D’une part, les développeurs peuvent décider de renoncer aux avantages et aux difficultés de la vente de leur logiciel et de le donner sans contrepartie. En anglais, ce logiciel est “gratuit” (*free*) dans le sens où personne ne doit le payer, mais il n’est gratuit que comme la bière que l’on offre. Cette distribution gratuite ne dit rien sur les autres conditions de licence et laisse à l’utilisateur l’obligation de toujours y regarder de plus près : Il se peut qu’il n’ait pas reçu les droits nécessaires pour modifier ou distribuer le logiciel, par exemple.

D’autre part, les développeurs peuvent choisir de faire de leur logiciel un “logiciel libre” (*free software*) au sens de Stallman. Ce terme (qu’il a inventé dans les années 1980) fait référence aux libertés essentielles de l’utilisateur, résumées comme suit :

1. Exécuter le logiciel
2. L’étudier
3. Le donner à d’autres (le *redistribuer*)
4. Redistribuer des copies des versions modifiées

Dans cette définition, le terme “logiciel libre” est défendu par la Free Software Foundation (FSF) qui a introduit le terme *copyleft* ou *gauche d’auteur* (qui n’a pas de signification juridique, mais présente une considération philosophique) pour caractériser la licence des logiciels libres.

Logiciels à code source ouvert (*open source*)

En dehors du mouvement du logiciel libre, les défenseurs des logiciels open source sont apparus à la fin des années 1990 comme un moyen de rendre les logiciels libres plus faciles à comprendre et plus populaires auprès des personnes extérieures au mouvement. Une fondation à but non lucratif, l'*Open Source Initiative* (OSI), a été créée en 1998, et Linus Torvalds, auteur initial du noyau Linux, a apporté son soutien au concept. L’Open Source Initiative a formalisé l'*Open Source Definition* (définition de l’open source) en y incluant les critères suivants :

1. *Redistribution libre* : Chacun est libre de décider comment redistribuer le programme, que ce soit gratuitement ou payant, tant qu’aucune redevance ou droit de licence n’est exigé. Cette liberté inclut l’incorporation du programme dans un autre programme.
2. *Disponibilité du code source*, qu’il soit en ligne ou fourni avec le logiciel.
3. Permettre la création et la distribution de *travaux dérivés* et de modifications.
4. *Intégrité du code source de l’auteur* : Les modifications peuvent être restreintes si les destinataires sont autorisés à modifier le programme au moyen de correctifs (*patches*) au moment de son développement et à distribuer ces correctifs en même temps que le code source. La distribution d’un logiciel *construit* à partir d’un code source modifié ne peut pas être restreinte.
5. *Pas de discrimination contre des personnes ou des groupes* : Par exemple, une licence autorisant l’utilisation du logiciel par les “enseignants uniquement” ne répondrait pas à l'*Open Source Definition*.
6. *Non-discrimination contre des champs d’application* : Par exemple, ne pas restreindre l’utilisation commerciale.
7. *Distribution de licence* : Toute personne qui reçoit le programme dispose de la même licence originale.
8. *La licence ne doit pas être spécifique à un produit*.
9. *La licence ne doit pas restreindre d’autres logiciels* : Par exemple, d’autres logiciels intégrés à ce logiciel peuvent avoir une licence différente.
10. *La licence doit être neutre sur le plan technologique*.

Liberté vs Open Source

La Free Software Foundation ne recommande pas le terme “open source”, insistant sur le fait qu’il cache l’objectif principal de la liberté. Ainsi, alors que les défenseurs des logiciels libres et des logiciels open source semblent poursuivre et défendre le même concept, les mouvements ont des motivations distinctes. Pour simplifier, les défenseurs des logiciels libres mettent l’accent sur les droits des développeurs et des utilisateurs, tandis que les défenseurs des logiciels open source donnent la priorité à la large utilisation et au succès du logiciel.

Pratiquement tous les logiciels libres sont considérés en tant qu’open source, alors qu’il existe de nombreuses licences considérées comme open source (approuvées par l’OSI), mais qui ne sont pas libres selon la FSF.

Étant donné que les différences entre le libre et l’open source concernent davantage les objectifs et les motivations que le contenu des licences, et que les deux termes restent fréquemment utilisés, de nombreux partisans se réfèrent aux deux définitions ensemble en utilisant les expressions *Free and Open Source Software* (FOSS) ou *Free/Libre and Open Source Software* (FLOSS). Le terme “libre” fait référence à la liberté dans de nombreuses langues romanes.

Autres types de logiciels gratuits

En plus des grandes catégories des logiciels FOSS et des logiciels propriétaires, il existe une série d’autres stratégies de distribution. Parmi ces stratégies, il y a :

Logiciel à contribution (*shareware*)

Ce terme fait généralement référence à un logiciel propriétaire qui est vendu, mais qui peut être utilisé gratuitement avec des fonctionnalités limitées jusqu’à ce que l’utilisateur décide d’acheter la version complète.

Logiciel gratuit (*freeware*)

Ce terme décrit un logiciel distribué gratuitement et sans limitation d’utilisation, mais pas nécessairement en conformité avec une définition formelle du logiciel libre. Dans de nombreux cas, les logiciels gratuits sont propriétaires et le code source n’est souvent pas publié du tout.

Logiciel à code source disponible

Il arrive que les développeurs de logiciels propriétaires mettent leur code source à disposition (pour faciliter les rapports de bogues, par exemple), mais imposent l’acquisition d’une licence propriétaire comme condition à l’utilisation du code source dans d’autres projets. Ce type de mise à disposition avec des limitations strictes ne doit pas être confondu avec le logiciel open source proprement dit.

Logiciel à code source partagé

Ce terme a été introduit par Microsoft en 2001, lorsque l'entreprise a décidé de mettre en ligne une partie du code source de ses logiciels à des fins de recherche et de test. Ne confondez pas cette définition étroite avec celle de *shareware* ou de logiciel à code source disponible.

Logiciel du domaine public

Il s'agit d'un logiciel pour lequel les auteurs ont renoncé à tous leurs droits. Cette définition ne s'applique pas dans toutes les juridictions (en particulier dans celles où l'auteur bénéficie du "droit d'auteur", comme en France ou en Allemagne). Des licences telles que l'"Unlicense" ont été introduites pour avoir le même effet. Par ailleurs, les logiciels peuvent également entrer dans le domaine public lorsque la durée du droit d'auteur (*copyright*) a expiré.

Principes de la législation sur les droits d'auteur et leur incidence sur les licences des logiciels open source

Premièrement et avant tout : si aucune information sur la licence n'est disponible pour un certain code source d'un fichier ou d'un projet, vous ne pouvez pas supposer que ce fichier ou ce projet n'est pas protégé par le droit d'auteur. En effet, c'est le contraire qui est vrai, du moins depuis que la plupart des pays du monde ont signé la *Convention de Berne* en 1887.

Dans ce traité, les nations signataires conviennent qu'une œuvre littéraire ou artistique est protégée par le droit d'auteur dès qu'elle existe (ou, en d'autres termes, dès qu'elle est "fixée" sur un support). Cela signifie qu'un auteur n'a pas besoin d'enregistrer ou de déposer une demande de droit d'auteur. Il peut encore le faire dans certains pays, et l'enregistrement peut être nécessaire pour les actions en contrefaçon dans certaines juridictions, y compris les États-Unis.

En outre, les signataires s'engagent à respecter les droits d'auteur de tout auteur d'un autre pays signataire, ce qui, en novembre 2022, représentait 181 des 195 pays du monde.

Cependant, tout ce qui a été créé n'est pas protégé par le droit d'auteur. Pour être considérée comme une œuvre protégée par le droit d'auteur, la création doit satisfaire à un certain nombre de critères de base : Par exemple, les faits et les idées ne peuvent pas être protégés par le droit d'auteur, mais un texte expliquant une idée peut être protégé s'il atteint un certain degré d'*originalité*. La mesure de l'originalité diffère d'un pays à l'autre, mais dans de nombreux cas, la barrière de protection est très faible. De nombreuses juridictions examinent actuellement dans quelle mesure l'intelligence artificielle peut être utilisée pour créer une œuvre méritant l'originalité et donc le droit d'auteur.

Selon la juridiction, les programmes informatiques sont protégés par le droit d'auteur en tant qu'œuvres littéraires : En d'autres termes, le droit d'auteur ne s'applique pas à l'idée ou à

l'algorithme, mais à sa transposition dans le code source.

Le droit d'auteur confère à l'auteur les droits exclusifs (entre autres) de copier, de modifier, d'accorder des sous-licences, de distribuer et de publier l'œuvre. La réception de l'œuvre est gratuite, de sorte qu'il n'est pas nécessaire d'obtenir une licence pour lire un livre ou écouter une chanson à la radio, tant qu'il n'est pas nécessaire d'en faire une copie permanente.

Étant donné que les droits d'auteur naissent sans qu'il soit nécessaire de les enregistrer, toute personne qui souhaite copier, modifier, accorder des sous-licences, distribuer ou publier l'œuvre d'un autre auteur doit d'abord en demander l'autorisation. C'est là que les licences entrent en jeu, en tant que contrats entre l'auteur d'une œuvre et une personne qui souhaite exercer certains des droits exclusifs de l'auteur.

Les licences FOSS sont proposées à tout le monde sans frais. Chacun peut créer sa propre licence et essayer de la faire approuver par l'OSI ou par la FSF. Mais l'utilisation d'une licence FOSS existante est fortement recommandée en raison de son acceptation générale et de la familiarité des utilisateurs avertis avec le contenu de la licence et ses obligations. En fait, seulement une poignée des nombreuses licences approuvées par la FSF ou l'OSI est couramment utilisée.

Principes du droit des brevets

Contrairement aux droits d'auteur, qui ne protègent pas les idées, les brevets protègent les inventions (idées) sans qu'il soit nécessaire que l'idée soit fixée (déjà) dans une machine ou un procédé. Une autre différence réside dans le fait que les inventeurs doivent déposer une demande de brevet explicite et l'enregistrer auprès de l'office des brevets du pays où la protection est demandée.

Sans plonger trop profondément dans le droit des brevets et ses exigences, nous nous contenterons de souligner un point essentiel : La protection par brevet exige (entre autres critères) une idée présentant un certain aspect technique. Historiquement, l'idée d'une nouvelle recette de cuisine ou d'un nouveau jeu de société ne remplit pas les conditions requises pour être protégée par un brevet, alors que l'idée d'une nouvelle machine à cuisiner ou d'une console de jeu peut remplir ces conditions.

Dans le contexte de la programmation informatique, la question se pose de savoir si les logiciels peuvent faire l'objet d'une protection par brevet. Cela dépend à la fois de la juridiction et de l'application particulière : En Allemagne, par exemple, les programmes informatiques en tant que tels sont généralement exclus de la protection par brevet. Toutefois, si les programmes sont combinés à un objet physique — par exemple, lorsqu'un logiciel contrôle un système de freinage automatisé dans une voiture — la protection par brevet peut être demandée pour l'application comprenant l'élément physique du frein et non le logiciel en tant que tel, et pourrait donc être

admissible.

Dans d'autres juridictions, telles que les États-Unis, des brevets peuvent être accordés pour des programmes informatiques en tant que tels, en fonction de l'évolution de la jurisprudence.

Le concept de protection des brevets doit être gardé à l'esprit dans les licences de logiciels libres. Certaines licences (comme la GPLv3) autorisent les brevets sur les logiciels libres en accordant explicitement la permission d'utiliser le logiciel. Mais certaines licences ne mentionnent même pas les brevets (comme la licence BSD-3-Clauses) et d'autres les excluent explicitement (comme les licences Creative Commons). Toutefois, dans certaines circonstances, l'octroi de brevets peut être implicite dans la licence ou peut être lu dans la licence.

NOTE Les différentes licences FOSS seront abordées dans les leçons suivantes.

En particulier, lorsqu'il s'agit de logiciels intégrés, tels que les logiciels dans les appareils audio, il convient d'accorder une attention particulière aux brevets liés au logiciel, par exemple en effectuant des vérifications auprès des auteurs du logiciel.

Contrats de licence

Comme indiqué précédemment, les licences sont des contrats entre l'auteur d'une œuvre et quelqu'un qui souhaite exercer certains des droits exclusifs de l'auteur. Certaines des licences FOSS les plus étendues, telles que les licences publiques générales GNU version 2 et version 3, comportent des dispositions relatives à la résiliation du contrat. Les licences FOSS comprennent toujours des concessions de droits concernant les libertés centrales. En général, les droits suivants sont énoncés ou peuvent être lus dans le texte de la licence :

...les droits d'utiliser, de copier, de modifier, de fusionner, de publier, de distribuer, d'accorder des sous-licences et/ou de vendre des copies du logiciel...

— MIT license, Traduction non officielle

Les *régisseurs de licence* (*license stewards*) sont des personnes ou, dans de nombreux cas, des organisations telles que la FSF, qui gèrent les versions des licences. Par exemple, la section 9 de la GPLv2 déclare que la FSF est le régisseur de la licence :

La Free Software Foundation peut publier de temps à autre des versions révisées et/ou nouvelles de la licence publique générale. Ces nouvelles versions seront similaires dans l'esprit à la présente version, mais pourront différer dans le détail pour répondre à de nouveaux problèmes ou préoccupations.

— GNU General Public License version 2, Traduction non officielle

Certains régisseurs de licence publient également une foire aux questions (FAQ) pour répondre aux questions concernant les licences. Ces questions peuvent être utiles pour engager la conversation entre le demandeur et le titulaire de la licence.

Distribution

La distribution de logiciels (sous forme binaire ou de code source, en particulier) est un aspect central de la plupart des licences FOSS, puisque la simple *utilisation* d'un logiciel FOSS ne déclenche généralement pas d'obligations en matière de licence :

Les activités autres que la copie, la distribution et la modification ne sont pas couvertes par la présente licence ; elles sortent de son champ d'application. L'exécution du programme n'est pas limitée...

— GNU General Public License version 2, Traduction non officielle

La plupart des obligations en matière de licence ne sont liées qu'à la distribution, c'est-à-dire à la transmission d'une copie modifiée ou non, que ce soit sur un support matériel tel qu'un CD ou par le biais d'un téléchargement. Dans certains cas, les conditions de licence sont également déclenchées si le logiciel est exécuté sur un serveur (c'est-à-dire sans jamais distribuer le code source) pendant que l'utilisateur interagit avec le logiciel.

La distribution d'un logiciel exécutable peut nécessiter la fourniture du code source, du texte de la licence et des avis de droits d'auteur, selon le type de licence FOSS. Certaines licences exigent que des avis de modification soient insérés dans le code source si un code modifié est distribué.

La distribution peut également avoir des effets sur le copyleft, c'est-à-dire que lors de la distribution d'un logiciel sous licence GPLv3 qui a été modifié, le code source de l'ensemble du logiciel modifié doit être publié sous la GPLv3.

Si les FOSS peuvent être vendus, il n'est généralement pas possible d'imposer des coûts de licence. Cela signifie, par exemple, que quelqu'un peut vendre un logiciel sous licence GPLv3 sous la forme d'un simple binaire, mais comme le code source est disponible (ou doit être offert), les personnes intéressées par le logiciel peuvent toujours choisir d'obtenir les sources (peut-être auprès de quelqu'un d'autre, gratuitement) et de construire le logiciel à partir des sources.

Travaux dérivés

Lorsqu'un groupe de développeurs incorpore dans son propre projet du code provenant d'un autre projet, le résultat peut être un travail *dérivé* ou *dérivant*. Les détails varient d'un projet à un autre et d'une licence à une autre, et comme ils nécessitent une certaine sophistication des techniques de développement logiciel, nous nous contenterons de dire que les développeurs

doivent s'assurer qu'ils incorporent le code d'une manière conforme à la licence.

La question la plus importante concernant les travaux dérivés est que, pour certaines licences telles que la GPL, un travail dérivé doit être publié sous la même licence. Ces licences sont appelées *récioproques*.

La signification pratique immédiate d'une exigence réciproque est que, si vous utilisez du code GPL dans votre propre projet d'une manière qui en fait un travail dérivé, vous devez révéler votre code source et permettre à d'autres de construire des produits à partir de ce code. Cette exigence de licence est fortement souhaitée par certains développeurs qui veulent encourager un plus grand nombre de personnes à utiliser des licences libres. Cependant, la licence réduit l'attrait de la GPL pour certains développeurs qui pourraient potentiellement utiliser du code libre.

De nombreuses autres licences libres et open source n'imposent pas cette exigence. On les appelle souvent des licences *permissive*.

Conséquences des violations de licence

Si, au moment de la distribution, les conditions de la licence ne sont pas respectées (par exemple, si un logiciel sous licence GPLv3 distribué sous forme de binaire n'est pas accompagné d'une offre de code source), le contrat de licence est violé. Les conséquences des violations de licence dépendent de la licence. La violation de la licence GPLv3, par exemple, peut entraîner la résiliation de la licence. Toute autre action nécessitant l'autorisation de l'auteur, par exemple la distribution du logiciel, constitue alors une violation du droit d'auteur.

Si une entreprise inclut un logiciel sous licence GPLv3 dans un produit et qu'elle viole la licence, une plainte peut être déposée pour obliger l'entreprise à rappeler ses produits. Les violations intentionnelles du droit d'auteur peuvent même donner lieu à des poursuites pénales.

Certaines licences contiennent des clauses spécifiques permettant au titulaire de la licence de remédier à la violation dans un délai de 30 jours à compter de la notification. Si la personne qui distribue le logiciel parvient à se conformer à la licence dans ce délai, la licence est rétablie.

Compatibilité et incompatibilité des licences

Les grands projets logiciels comprennent souvent des logiciels sous différentes licences, chaque licence spécifiant ses propres exigences. Ces projets peuvent se heurter à des obstacles lors de l'utilisation d'un logiciel dont la licence doit être utilisée pour des travaux dérivés. En effet, les conditions d'utilisation des licences copyleft peuvent être différentes, ce qui les rend incompatibles. La publication ou la distribution d'un projet logiciel qui intègre des composants sous différentes licences copyleft peut ne pas être possible sans violer l'une des licences.

Certaines licences copyleft énumèrent explicitement les licences compatibles, ce qui facilite l'utilisation d'un composant sous licence copyleft dans le cadre d'une autre licence copyleft.

La plupart des licences permissives sont compatibles avec d'autres licences. Par exemple, les composants sous licence MIT peuvent être utilisés dans un projet sous licence GPLv3 sans risquer de violer la licence. Toutefois, un composant sous licence GPLv3 ne peut pas toujours être utilisé dans un projet sous licence MIT sans violer les termes de la GPLv3. La compatibilité ne fonctionne donc pas toujours dans les deux sens.

Avant qu'un projet logiciel ne soit mis sur le marché, les développeurs et leurs conseillers juridiques devraient procéder à un contrôle approfondi de la compatibilité des licences, afin d'éviter toute violation de celles-ci. La gestion de la conformité des logiciels open source doit être intégrée dès les premières étapes du processus de développement d'un logiciel pour éviter les retards dans la distribution du logiciel causés, par exemple, par des problèmes d'incompatibilité de licence. Recherchez dans le code source les licences applicables (par exemple, en utilisant des outils d'analyse de logiciels) et vérifiez si toutes les conditions de licence sont remplies.

Double licence et licences multiples

Certains logiciels peuvent être disponibles sous plusieurs licences. Par exemple, un donneur de licence peut choisir d'accorder une double licence pour son projet, à la fois sous une licence copyleft telle que la GPLv3 et sous une licence propriétaire. La licence propriétaire peut être requise, par exemple, si le preneur de licence potentiel incorpore le code dans son propre produit propriétaire. Chaque développeur détermine s'il peut adhérer aux conditions de la GPLv3 ou s'il doit obtenir la licence propriétaire, ce qui entraînera très probablement des coûts de licence.

Comme indiqué précédemment, certains logiciels peuvent inclure des composants sous diverses licences avec des conditions de licence différentes. Bien que la plupart des licences n'entraînent pas souvent d'incompatibilité, les différentes licences peuvent être assorties d'exigences différentes pour les différentes parties du logiciel.

Exercices guidés

1. Que signifie l'acronyme FOSS ?

2. Lesquels des éléments suivants font explicitement partie des libertés essentielles de l'utilisateur d'un logiciel libre ?

Exécuter le logiciel	
Étudier le logiciel	
Copier le logiciel	
Modifier le logiciel	
Publier le logiciel	
Redistribuer le logiciel	

3. Le code source trouvé sur l'internet sans information sur les licences peut-il être modifié et distribué librement par n'importe qui ? Expliquez pourquoi.

4. Les logiciels peuvent-ils être brevetés ?

Oui	
Non	
Cela dépend	

5. Qu'est-ce qu'un régisseur de licence (*license steward*) ?

La même chose que le donneur de licence	
Quelqu'un qui peut proposer des versions futures d'une licence	
Quelqu'un qui peut mettre fin à une licence	
Quelqu'un qui gère les logiciels des avions	

6. La compatibilité des licences fonctionne-t-elle toujours dans les deux sens ?

Oui, si deux licences sont compatibles, il importe peu que A soit inclus dans B ou que B soit inclus dans A.

Non, dans certains cas, la licence A peut-être incluse dans un projet sous licence B, mais la licence B ne peut pas être distribuée sous la licence A.

Non, la compatibilité des licences est toujours unidirectionnelle.

Exercices d'exploration

1. Les licences GPLv2 et LGPLv2.1 sont-elles compatibles ? Veuillez donner une brève explication.

2. Les logiciels peuvent-ils être publiés sous des licences de contenu ouvert (telles que CC-BY) ?

Oui, les licences CC peuvent être appliquées à toute œuvre protégée par le droit d'auteur.	
Non, les logiciels ne peuvent être protégés que par des brevets.	
Non, les licences CC ne s'appliquent pas aux logiciels.	

3. Pourquoi la distribution est-elle importante dans le contexte des licences FOSS ?

Résumé

Cette leçon présente les concepts de base des logiciels libres et open source ainsi que les concepts sous-jacents du droit d'auteur et des brevets. Elle explique certains des fondements et de l'histoire des logiciels libres et des logiciels open source et aide à les distinguer des concepts de licences propriétaires. L'*Open Source Definition* par l'OSI permet de classer les licences dans la catégorie des licences open source.

Bien qu'il existe déjà des dizaines de licences FOSS, chacun est libre de proposer des licences supplémentaires.

Le concept de licence ne doit pas être sous-estimé : Les licences autorisent l'utilisation d'un logiciel d'une manière qui serait autrement réservée exclusivement à l'auteur. Les violations de licences peuvent donner lieu à des litiges juridiques. Il convient donc de demander un avis juridique sur la conformité des licences avant de distribuer un logiciel ou de l'incorporer dans le code d'un autre projet.

Réponses aux exercices guidés

1. Que signifie l'acronyme FOSS ?

Logiciels libres et open source (*Free and Open Source Software*)

2. Lesquels des éléments suivants font explicitement partie des libertés essentielles de l'utilisateur d'un logiciel libre ?

Exécuter le logiciel	X
Étudier le logiciel	X
Copier le logiciel	
Modifier le logiciel	X
Publier le logiciel	
Redistribuer le logiciel	X

3. Le code source trouvé sur l'internet sans information sur les licences peut-il être modifié et distribué librement par n'importe qui ? Expliquez pourquoi.

Non. Le code source, s'il atteint le seuil d'originalité, est par défaut protégé par le droit d'auteur en tant qu'œuvre littéraire. Comme la modification et la distribution constituent des droits exclusifs de l'auteur, personne d'autre que l'auteur ne peut le faire ou donner l'autorisation de le faire.

4. Les logiciels peuvent-ils être brevetés ?

Oui	
Non	
Cela dépend	X

5. Qu'est-ce qu'un régisseur de licence (*license steward*) ?

La même chose que le donneur de licence	
Quelqu'un qui peut proposer des versions futures d'une licence	X
Quelqu'un qui peut mettre fin à une licence	
Quelqu'un qui gère les logiciels des avions	

6. La compatibilité des licences fonctionne-t-elle toujours dans les deux sens ?

Oui, si deux licences sont compatibles, il importe peu que A soit inclus dans B ou que B soit inclus dans A.	
Non, dans certains cas, la licence A peut-être incluse dans un projet sous licence B, mais la licence B ne peut pas être distribuée sous la licence A.	X
Non, la compatibilité des licences est toujours unidirectionnelle.	

Réponses aux exercices d'exploration

1. Les licences GPLv2 et LGPLv2.1 sont-elles compatibles ? Veuillez donner une brève explication.

Si le logiciel A est sous licence GPLv2 et le logiciel B sous licence LGPLv2.1, il est possible d'utiliser B dans A, puisque la LGPLv2.1 permet une utilisation sous les conditions de la licence GPLv2. Cependant, A ne peut pas être utilisé dans B selon les termes de la LGPLv2.1. Si A est utilisé dans B, l'ensemble du logiciel doit être placé sous licence GPLv2, ce qui est possible, car la LGPLv2.1 autorise l'utilisation du logiciel sous la GPLv2.0.

2. Les logiciels peuvent-ils être publiés sous des licences de contenu ouvert (telles que CC-BY) ?

Oui, les licences CC peuvent être appliquées à toute œuvre protégée par le droit d'auteur.	X
Non, les logiciels ne peuvent être protégés que par des brevets.	
Non, les licences CC ne s'appliquent pas aux logiciels.	

3. Pourquoi la distribution est-elle importante dans le contexte des licences FOSS ?

De nombreuses obligations liées aux licences FOSS sont déclenchées lors de la distribution du logiciel. Si le logiciel n'est jamais distribué, mais seulement modifié et utilisé dans un système fermé (tel qu'une division d'une entreprise), il peut être possible d'utiliser le logiciel sans adhérer aux obligations de licence.



052.2 Licences de logiciels copyleft

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 052.2

Valeur

3

Domaines de connaissance les plus importants

- Comprendre le concept de licence de logiciel copyleft
- Comprendre les droits accordés par les licences de logiciels copyleft
- Comprendre les obligations créées par les licences de logiciels copyleft
- Comprendre les principales propriétés des principales licences de logiciels copyleft
- Comprendre la compatibilité des licences de logiciels copyleft avec d'autres licences de logiciels
- Connaissance des termes "licence réciproque" et "licence restrictive"

Liste partielle de termes, fichiers et utilitaires utilisés

- Copyleft
- Distribution
- Transport (Conveying)
- Tivoïsation
- Licence publique générale GNU, version 2.0 (GPLv2)
- Licence publique générale GNU, version 3.0 (GPLv3)
- Licence publique générale limitée GNU, version 2 (LGPLv2)
- Licence publique générale limitée GNU, version 3 (LGPLv3)

- Licence publique générale GNU Affero, version 3 (AGPLv3)
- Licence publique Eclipse (EPL), version 1.0
- Licence publique Eclipse (EPL), version 2.0
- Licence publique Mozilla (MPL)



Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	052 Licences de logiciels à code source ouvert
Objectif :	052.2 Licences de logiciels copyleft
Leçon :	1 sur 1

Introduction

L'importance des licences pour l'utilisation et le développement des logiciels a déjà été décrite. Il n'est donc pas surprenant que les logiciels libres aient été caractérisés dès le départ par de nouvelles approches en matière de licences : Les conditions d'utilisation sans restriction ou de développement collaboratif des logiciels doivent être définies légalement afin d'être protégées et appliquées.

Conscients que le droit d'auteur, solidement ancré dans la quasi-totalité des systèmes juridiques du monde, ne pouvait être ni remis en cause ni remplacé, les développeurs de logiciels ont adopté, dès le début des années 1980, une approche qui respecte les règles du droit d'auteur tout en les complétant par de nouvelles règles qui mettent l'accent sur le principe de la "liberté" : le *copyleft*.

Copyleft et la licence publique générale GNU

Richard Stallman, alors développeur au prestigieux MIT, a fondé le *projet GNU* en 1983 pour développer ce qu'il considérait comme un système d'exploitation "libre". Il est rapidement apparu que le code développé par le projet devait être légalement protégé afin qu'il ne puisse pas être simplement repris par des prestataires commerciaux et devenir ainsi "non libre".

Stallman a donc fondé en 1985 la *Free Software Foundation* (FSF), une organisation à but non lucratif, qui résume sa mission sur son site web comme suit : “La Free Software Foundation s’efforce de garantir la liberté des utilisateurs d’ordinateurs en encourageant le développement et l’utilisation de logiciels et de documentation libres...”.

Un instrument clé pour cette mission est une licence qui respecte la loi applicable (en particulier la loi sur les droits d’auteur) d’une part, et qui met en œuvre ses propres idées de liberté d’une manière juridiquement propre d’autre part. Le résultat a été la première version de la *licence publique générale GNU* (*GNU General Public License* - GPLv1) en 1989. Cette licence et de nombreux articles — tels que “What is Free Software?”, écrit par Stallman en 1992 — exposent clairement les motivations et les valeurs des développeurs de logiciels libres, qui se considèrent désormais comme un “mouvement”.

Le noyau programmatique est toujours formé par les “quatre libertés essentielles” formulées par Stallman dans l’article susmentionné, dont la numérotation commence par 0 :

- La liberté de faire fonctionner le programme comme vous voulez, pour n’importe quel usage (liberté 0).
- La liberté d’étudier le fonctionnement du programme, et de le modifier pour qu’il effectue vos tâches informatiques comme vous le souhaitez (liberté 1). L’accès au code source est une condition nécessaire.
- La liberté de redistribuer des copies, donc d’aider les autres (liberté 2).
- La liberté de distribuer aux autres des copies de vos versions modifiées (liberté 3). En faisant cela, vous donnez à toute la communauté la possibilité de profiter de vos changements. L’accès au code source est une condition nécessaire.

— Richard Stallman, What is Free Software? (traduction en français sur gnu.org)

Contrairement aux licences des produits commerciaux, qui placent les restrictions d’utilisation au premier plan, les logiciels libres offrent un maximum de liberté aux utilisateurs et aux développeurs.

Le préambule de la GNU GPLv1 résume cela comme suit :

Plus précisément, la licence publique générale est conçue pour garantir que vous avez la liberté de donner ou de vendre des copies de logiciels libres, que vous recevez le code source ou que vous pouvez l’obtenir si vous le souhaitez, que vous pouvez modifier le logiciel ou en utiliser des parties dans de nouveaux programmes libres, et que vous savez que vous pouvez faire ces choses.

— GNU General Public License version 1, Traduction non officielle

Cela signifie que chacun a le droit d'utiliser, de distribuer et de modifier un logiciel sous GPL sans restriction (ce qui est possible parce que le code source est accessible, c'est-à-dire "ouvert") et de distribuer à son tour les modifications. Il est même possible de demander de l'argent pour transmettre le logiciel :

Actuellement, nous encourageons les personnes qui redistribuent des logiciels libres à les facturer autant qu'elles le souhaitent ou qu'elles le peuvent. Si une licence ne permet pas aux utilisateurs de faire des copies et de les vendre, il s'agit d'une licence non libre... Les programmes libres sont parfois distribués gratuitement, et parfois pour un prix élevé. Souvent, le même programme est disponible dans les deux cas à des endroits différents. Le programme est libre quel que soit son prix, car les utilisateurs sont libres de l'utiliser.

— Free Software Foundation, *Selling Free Software*

Mais si les libertés sont si étendues, dans quelle mesure les logiciels sont-ils protégés par cette licence, par exemple contre l'incorporation dans des produits propriétaires ?

C'est le rôle du principe du copyleft, mentionné précédemment, que la GPL applique déjà dans la version 1 même si le terme n'apparaît pas encore explicitement :

Vous ne pouvez pas copier, modifier, concéder en sous-licence, distribuer ou transférer le programme, sauf dans les cas expressément prévus par la présente licence publique générale.

— GNU General Public License version 1, Traduction non officielle

Cela signifie que toutes ces libertés sont liées à la condition que les utilisateurs préservent ces libertés dans tout ce qu'ils font avec le logiciel.

Le copyleft ne se contente donc pas de garantir les libertés, mais exige également de tous les utilisateurs qu'ils accordent ces libertés aux autres. Pour ce faire, il est stipulé qu'un logiciel sous licence copyleft (comme la première GPLv1) ne peut être modifié et redistribué que si les modifications sont publiées dans les mêmes conditions, c'est-à-dire sous la même licence.

L'idéal du logiciel libre, à savoir l'utilisation collective et le développement continu du logiciel, prime donc sur les besoins personnels que les individus peuvent avoir par rapport au logiciel. Le principe de réciprocité est crucial : Ceux qui bénéficient des libertés doivent aussi les accorder. C'est pourquoi les licences copyleft sont souvent qualifiées de *réiproques*.

Cette approche totalement nouvelle d'une licence de logiciel s'est déjà avérée juridiquement solide et praticable dans la version 1 de la GPL, de sorte que la GPL n'a subi que deux révisions majeures au cours des quelque 40 années pendant lesquelles le marché moderne des technologies de l'information s'est développé.

GPLv2 et GPLv3

En 1991, la Free Software Foundation a présenté la version 2 de la licence publique générale GNU (GPLv2), qui s’est imposée pendant plusieurs années comme la licence la plus populaire pour les projets de logiciels libres. Par exemple, le cœur du système d’exploitation Linux est toujours sous la licence GPLv2 aujourd’hui.

Par rapport à la version 1, la version 2 s’attache principalement à donner des définitions plus précises afin d’éviter les ambiguïtés. Par exemple, la version 2 explique de manière beaucoup plus détaillée ce que l’on entend par “code source”.

La nouvelle section 7 est également intéressante. Elle pose le principe de la liberté et donc de la validité de la licence comme étant absolue et n’autorise aucun compromis — par exemple, l’intégration de parties moins libres dans le logiciel :

Si vous ne pouvez pas distribuer le programme de manière à satisfaire simultanément vos obligations au titre de la présente licence et toute autre obligation pertinente, vous pouvez en conséquence ne pas le distribuer du tout. Par exemple, si une licence d’un brevet ne permet pas la redistribution libre de droits du programme par tous ceux qui en reçoivent des copies directement ou indirectement par votre intermédiaire, la seule façon de satisfaire à la fois à cette licence et à d’autres obligations serait de s’abstenir complètement de distribuer le programme.

— GNU General Public License version 2, Traduction non officielle

Ce n’est que 16 ans plus tard, en 2007, que la FSF a publié une nouvelle version de la GPL afin de prendre en compte les innovations techniques — telles que la fourniture de services logiciels via l’internet — ainsi que les questions de compatibilité avec d’autres licences de logiciels libres. Toutefois, la licence reste stable en ce qui concerne ses déclarations fondamentales et se contente d’ajouter des détails pour plus de clarté. Examinons de plus près certains de ces ajouts.

Alors que la GPLv2 fait encore généralement référence à la mise à disposition de logiciels en tant que *distribution*, la GPLv3 précise ce processus à l’aide de deux nouveaux termes : *propagation* et *transmission*. La raison principale en est que le terme “distribution” est défini dans de nombreuses lois sur le droit d’auteur dans le monde entier. Pour éviter toute ambiguïté ou conflit, la GPLv3 choisit ces nouveaux termes et les définit comme suit :

“Propager” une œuvre signifie en faire quoi que ce soit qui, sans autorisation, vous rendrait directement ou indirectement responsable d’une infraction à la législation applicable en matière de droits d’auteur, à l’exception de l’exécution de l’œuvre sur un ordinateur ou de la modification d’une copie privée. La propagation comprend la copie, la distribution (avec ou sans modification), la mise à disposition du public et, dans certains pays, également d’autres

activités.

“Transmettre” une œuvre signifie toute forme de propagation qui permet à d’autres parties de faire ou de recevoir des copies. Une simple interaction avec un utilisateur par le biais d’un réseau informatique, sans transfert d’une copie, n’est pas une transmission.

— GNU General Public License version 3, Traduction non officielle

Face au nombre croissant de logiciels commerciaux dont la distribution est restreinte par les fabricants au moyen de mesures techniques telles que des codes d’enregistrement ou des composants matériels (appelés *dongles*), un certain nombre d’initiatives juridiques internationales ont été prises à la fin des années 1990 pour criminaliser le contournement de ces mesures. Cette *gestion des droits numériques* (DRM pour *digital rights management*), que les opposants appellent aussi *gestion numérique des restrictions* (*digital restrictions management*), est une épine dans le pied de la FSF, car ces mesures vont fondamentalement à l’encontre de l’exigence de la libre distribution des logiciels.

En réponse, la version 3 de la GPL contient un passage indiquant que les logiciels sous GPL ne peuvent pas être modifiés en fonction des exigences légales en matière de DRM. Cela signifie également que les logiciels sous licence GPLv3 peuvent utiliser des DRM, mais que d’autres sont également autorisés à contourner ces mesures.

Le terme *tivoïsation* est également fréquemment utilisé dans ce contexte. Ce terme est apparu explicitement dans les premières versions de la GPLv3, mais n’a pas été inclus dans la version finale. Le terme remonte à la société TiVo, qui utilisait des logiciels sous licence GPLv2 dans son enregistreur vidéo numérique, mais qui, en même temps, empêchait techniquement l’installation et l’utilisation de logiciels modifiés sur l’appareil. Selon la FSF, cela contredisait les principes de la GPL et, après quelques discussions, la GPLv3 en tient compte dans un paragraphe sur les *produits utilisateurs*. Elle stipule généralement que les produits qui utilisent des logiciels sous licence GPLv3 doivent aussi fournir des informations sur la manière dont ces logiciels peuvent être modifiés.

Un autre ajout concerne les *brevets*, que la FSF rejette fondamentalement dans les logiciels en raison de leur obstruction à la liberté et à l’innovation. Cela est déjà indiqué dans le préambule de la GPLv3 :

Enfin, chaque programme est constamment menacé par les brevets logiciels. Les États ne devraient pas autoriser les brevets à restreindre le développement et l’utilisation des logiciels sur les ordinateurs à usage général, mais dans ceux qui le font, nous souhaitons éviter le danger particulier que les brevets appliqués à un programme libre puissent le rendre effectivement propriétaire. Pour éviter cela, la GPL garantit que les brevets ne peuvent pas être utilisés pour rendre le programme non libre.

— GNU General Public License version 3, Traduction non officielle

Le texte de la licence contient également plusieurs passages qui autorisent l'inclusion de code sous un brevet au moyen d'une "licence de brevet non exclusive, mondiale et libre de redevance" du propriétaire de la licence afin de protéger les utilisateurs de ce code contre les litiges entre les détenteurs de brevets et les détenteurs de licences.

Licence publique générale GNU Affero

Avec l'augmentation de la disponibilité et de la vitesse de l'internet, de plus en plus de services apparaissent dans lesquels le logiciel est simplement installé sur les serveurs du vendeur — le *Fournisseur de services d'applications* (ASP pour *Application Service Provider*) — et dont les clients interagissent avec les services par l'intermédiaire de l'internet. Cette tendance a été baptisée *logiciel en tant que service* (SaaS pour *Software as a Service*).

Dans de tels cas, la GPLv2 n'indiquait pas clairement si et comment le code source (éventuellement modifié par le fournisseur) devait être mis à disposition. La version 3 de la GPL comble cette lacune, connue sous le nom de *brèche ASP* (*ASP loophole*), en faisant explicitement référence, dans la section 13, à une autre licence publiée par la FSF en 2007 : la *Licence publique générale GNU Affero* version 3 (*GNU Affero General Public License* version 3 - GNU AGPLv3). Le nom fait référence à la société Affero, qui a développé et publié les deux premières versions de cette licence.

Cette AGPLv3 correspond fondamentalement à la GPLv3, mais régit explicitement le problème de l'ASP dans la section "interaction réseau à distance". En outre, les deux licences indiquent explicitement qu'elles peuvent être combinées l'une avec l'autre sans restriction.

En résumé, la GNU AGPL est un complément à la GPL. L'AGPL étend le champ d'application de la GPL en appliquant le copyleft aux logiciels qui ne sont plus utilisés dans des installations locales, mais exclusivement sous la forme de services transmis via des réseaux.

Compatibilité des licences copyleft

Le développement de logiciels libres repose sur le travail d'autrui, c'est-à-dire sur l'intégration, la modification et le partage du code source d'autrui. Si toutes les parties d'un logiciel modifié ou nouvellement compilé sont soumises à la même licence copyleft, par exemple la GPLv3, cela est possible sans aucun problème juridique. La licence exige simplement que le résultat soit distribué sous la même licence.

Les choses se compliquent lorsque le logiciel se compose d'éléments faisant l'objet de licences différentes. Plusieurs facteurs doivent alors être pris en compte.

Œuvres combinées et dérivées

Les logiciels libres sont parfois créés dans des conditions très différentes. Les modifications vont de simples corrections d'erreurs à des projets complexes comportant des millions de lignes de code. Quelle que soit l'ampleur du projet, une distinction fondamentale est faite entre deux types d'œuvres lorsqu'il s'agit d'accorder des licences : *dérivées* et *combinées*.

Supposons, par exemple, qu'il manque une certaine fonctionnalité au projet logiciel A. Au lieu de développer cette fonctionnalité à partir de zéro, il est logique de combiner le code d'un autre projet B, qui offre exactement cette fonctionnalité, avec A. Le logiciel de B ne devrait même pas être modifié pour cela, mais simplement ajouté à A. Il s'agit donc d'une œuvre combinée. Si A et B sont soumis à la même licence copyleft, il n'y a aucun problème pour l'œuvre combinée.

Si A et B sont soumis à des licences copyleft différentes, il convient d'être prudent : La combinaison de A et B constitue-t-elle déjà une œuvre distincte ? Et si oui, sous quelle licence peut-elle ou doit-elle être placée ? Ou bien peut-on éviter un conflit en s'assurant que les deux parties A et B restent séparées avec leurs licences respectives et ne constituent pas une nouvelle œuvre ?

Cela devient encore plus difficile avec les œuvres dérivées, c'est-à-dire lorsque le projet A ne peut utiliser la fonctionnalité de B qu'en incorporant le code de B directement dans le code de A. Cette intégration crée une nouvelle œuvre dérivée dont les parties ne peuvent plus être séparées.

Le copyleft entre en jeu dans le cas d'une œuvre dérivée. Si, par exemple, A est sous une licence copyleft telle que la GPLv3, la nouvelle œuvre dérivée doit également être sous la GPLv3 conformément au principe de réciprocité. Mais qu'en est-il si B est soumis à une licence différente ? S'agit-il d'une licence copyleft, et est-elle compatible avec la GPLv3 ? Ou, s'il s'agit d'un type de licence différent, B peut-il aussi être publié sous une licence différente, c'est-à-dire sous une nouvelle licence ? Ou encore, les licences A et B s'excluent-elles mutuellement de manière catégorique ?

L'objectif de cette leçon n'est pas d'énumérer les combinaisons possibles et les solutions envisageables. L'important est d'illustrer la complexité du problème résultant de la combinaison des différentes situations.

D'un point de vue *technique*, la première chose à clarifier est la manière dont les différentes parties du logiciel (dans notre exemple A et B) fonctionnent ensemble : Peuvent-elles être séparées l'une de l'autre ou existe-t-il des processus dont l'exécution ne peut plus être clairement attribuée à l'une ou l'autre partie ?

D'un point de vue juridique, des questions se posent quant à la relation entre les licences de A et de B. Sont-elles compatibles entre elles, ou seulement en partie, ou uniquement dans un sens ? Le changement de licence est-il une option ?

Nous ne pouvons ici que souligner la complexité de ces questions, et non les résoudre. De telles décisions requièrent de solides connaissances juridiques. Ainsi, les décisions relatives à l'octroi de licences pour les nouvelles œuvres, mais surtout pour les œuvres combinées et dérivées, doivent être prises *précocement*, *prudemment*, et toujours après un *conseil juridique* détaillé.

Copyleft faible

Le copyleft s'est avéré extrêmement robuste au cours des dernières décennies, en particulier sous la forme de la GPL dans ses versions 2 et 3. Cependant, des exigences techniques particulières ont incité la FSF à réagir en adaptant ses licences. La GNU Affero General Public License en est un exemple. D'autres exemples sont présentés dans les sous-sections suivantes.

Licence publique générale GNU amoindrie

Une méthode largement utilisée dans le développement de logiciels est l'utilisation de petits modules pour les tâches standard, telles que l'ouverture ou l'enregistrement de fichiers. Ces modules — ou collections de modules — sont appelés *bibliothèques logicielles*. Il ne s'agit généralement pas d'applications exécutables de manière indépendante, mais de routines que l'application intègre en fonction des besoins. Le processus d'intégration est connu sous le nom de *édition de liens* ou *linking*, ce qui permet de distinguer deux méthodes.

Avec les *bibliothèques statiques*, l'application (via des programmes auxiliaires et des étapes intermédiaires) intègre fermement le code des modules dans le fichier exécutable de l'application. Avec les *bibliothèques dynamiques*, en revanche, l'application n'intègre un module que lorsqu'il est nécessaire au moment de l'exécution et le charge dans la mémoire vive.

Cela soulève une question concernant le copyleft et les licences : Quelles sont les implications de la liaison en matière de licence ? Par exemple, cette forme de reprise du code exige-t-elle qu'une application qui utilise une bibliothèque sous GPL adopte elle-même la GPL ? Et cela s'applique-t-il à la fois à la liaison statique et à la liaison dynamique ?

Le processus d'édition de liens est tellement omniprésent dans le développement de logiciels, et les problèmes associés au copyleft réciproque sont tellement importants, que la FSF a cherché très tôt une solution de licence qui autorise l'édition de liens par le biais de la *Licence publique générale GNU amoindrie* (*GNU Lesser General Public License* - LGPL). La LGPL est mise à jour en même temps que chaque nouvelle version de la GPL. À l'origine, la licence s'appelait *Licence publique générale GNU pour les bibliothèques* (*GNU Library General Public License*), ce qui révélait le problème initial à l'origine de la création de la licence. Dans la version 3, "bibliothèque" a été remplacé par "amoindrie" pour indiquer ce qui est réellement en jeu, à savoir un affaiblissement pragmatique du principe du copyleft.

Une bibliothèque sous licence LGPL peut donc être utilisée par un logiciel sans que ce logiciel soit lui-même automatiquement soumis au copyleft. Les projets de logiciels sous licences dites *permissives* (abordées dans une autre leçon), en particulier, bénéficient de ce compromis, car il crée une protection juridique pour la combinaison d’approches qui ne sont pas en soi compatibles en vertu du droit des licences.

Toute modification apportée à un logiciel sous licence LGPL reste soumise au copyleft, c’est-à-dire qu’elle doit également être placée sous LGPL.

Toutefois, la bibliothèque sous licence LGPL doit être interchangeable, afin de permettre à l’utilisateur du logiciel de remplacer la bibliothèque par une version modifiée. Des conditions appropriées doivent être créées à cet effet, c’est-à-dire que des informations doivent être fournies sur la manière dont un tel remplacement peut être effectué.

Autres licences copyleft

D’autres projets et organisations de logiciels libres recherchent également le cadre juridique le mieux adapté à leurs besoins et développent donc leurs propres licences. C’est le cas de la *Fondation Mozilla*, fondée en 1998 et aujourd’hui surtout connue pour les deux projets qu’elle soutient : le navigateur internet Firefox et le client de messagerie Thunderbird.

La version 1 de la *Licence publique Mozilla* (*Mozilla Public License* - MPL) a été publiée en 1998, et la version actuelle (en 2024) 2.0 en 2012.

Comme la LGPL, la MPL est souvent considérée comme une licence “copyleft faible”. En fait, elle cherche à trouver un équilibre entre les exigences strictes du copyleft et les possibilités d’intégration avec des produits commerciaux. Elle y parvient, entre autres, grâce à un principe connu sous le nom de *copyleft au niveau fichier* (*file-level copyleft*) : Si vous modifiez un fichier appartenant à un logiciel sous MPL, vous pouvez intégrer ce fichier dans un logiciel propriétaire tant que le fichier modifié lui-même reste sous MPL et il est donc accessible.

Un autre exemple de licence copyleft faible est la *Licence publique Eclipse* (*Eclipse Public License* - EPL) de la Fondation Eclipse. La version actuelle 2.0 de 2017 est très similaire à la MPL et elle est souvent considérée comme la licence copyleft la plus “favorable aux entreprises”. Cependant, les différentes licences copyleft — et il y en a d’autres en plus de celles mentionnées ici — sont généralement nées de développements historiques plutôt que de différences juridiques claires.

Exercices guidés

1. Quelle est la signification de l'abréviation GPL ?

2. Pourquoi les licences copyleft sont-elles également appelées réciproques ?

3. Quelle licence copyleft de la FSF est adaptée aux bibliothèques logicielles ?

4. Quels termes remplacent le terme “distribution” dans la GPLv3 et pourquoi ?

5. Parmi les licences copyleft suivantes, lesquelles ont été émises par la Free Software Foundation ?

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

Exercices d'exploration

1. Parmi les licences suivantes, lesquelles sont des licences copyleft ?

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

2. Est-il possible de créer une œuvre dérivée combinant des parties de deux projets de logiciels qui sont soumis à des différentes licences copyleft fortes ? Justifiez votre réponse !

3. Parmi les licences suivantes, lesquelles ont un copyleft fort et lesquelles ont un copyleft faible ?

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

4. Décrivez quelques problèmes de compatibilité qui pourraient survenir lors de la combinaison d'un logiciel sous une licence copyleft faible avec un logiciel sous une licence non copyleft.

Résumé

Cette leçon traite des caractéristiques des licences de logiciels qui suivent le principe du copyleft. Développée dans les années 1980 par la Free Software Foundation, la GNU General Public License (GPL) est actuellement la licence la plus populaire avec un droit d'auteur fort. Malgré plusieurs révisions jusqu'à la version 3 actuelle, ses exigences de base sont restées pratiquement inchangées : les libertés accordées par la licence d'utiliser, de redistribuer et de modifier le logiciel sans restriction doivent être préservées à tout moment. Cela signifie que le logiciel modifié ne peut être distribué que sous les mêmes conditions (c'est-à-dire la même licence).

Les innovations techniques ainsi que le désir de disposer d'une marge de manœuvre juridique pour collaborer avec d'autres projets ont incité la FSF à développer des licences supplémentaires ou alternatives. La GNU Lesser General Public License (LGPL), par exemple, prend en compte la liaison statique ou dynamique de bibliothèques logicielles fréquemment utilisées dans le développement de logiciels. La GNU Affero General Public License (AGPL) répond à la tendance technique qui consiste à utiliser les logiciels sous forme de services via le réseau (en particulier l'internet), c'est-à-dire pas dans des installations locales.

Outre la FSF, d'autres projets tels que la Fondation Mozilla et la Fondation Eclipse ont développé des licences copyleft. Ceux-ci recherchent également un compromis entre la garantie des libertés accordées par la licence et une relation plus simple avec les logiciels sous d'autres licences au moyen d'un copyleft plus faible.

En principe, la compatibilité des licences est une question importante pour les licences copyleft, car le principe du développement libre et collaboratif de logiciels doit être concilié avec les conditions juridiques déterminées par la licence respective. Dans toute forme de combinaison de logiciels sous différentes licences, les conditions juridiques doivent être examinées attentivement dans chaque cas.

Réponses aux exercices guidés

1. Quelle est la signification de l'abréviation GPL ?

General Public License

2. Pourquoi les licences copyleft sont-elles également appelées réciproques ?

Le principe du copyleft exige que les libertés accordées par une licence soient également accordées à d'autres sans restriction. Par exemple, si vous apportez une modification à un logiciel sous GPL, vous êtes tenu de mettre ces modifications à la disposition d'autres personnes dans les mêmes conditions, conformément à ce principe de réciprocité.

3. Quelle licence copyleft de la FSF est adaptée aux bibliothèques logicielles ?

GNU Lesser General Public License (GNU LGPL)

4. Quels termes remplacent le terme "distribution" dans la GPLv3 et pourquoi ?

Les termes "transmettre" et "propager" remplacent le terme "distribution". La raison en est que le terme "distribution" est fermement ancré dans la loi internationale sur le droit d'auteur. Pour éviter les conflits ou les malentendus, la GPL dans sa version 3 n'utilise pas ce terme.

5. Parmi les licences copyleft suivantes, lesquelles ont été émises par la Free Software Foundation ?

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

Réponses aux exercices d'exploration

1. Parmi les licences suivantes, lesquelles sont des licences copyleft ?

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

2. Est-il possible de créer une œuvre dérivée combinant des parties de deux projets de logiciels qui sont soumis à des différentes licences copyleft fortes ? Justifiez votre réponse !

Les licences avec un copyleft fort exigent généralement que les versions modifiées soient sous la même licence. Cela exclut également l'octroi de nouvelles licences. La combinaison de licences, lorsque les deux suivent ces principes, représente donc une contradiction insoluble.

3. Parmi les licences suivantes, lesquelles ont un copyleft fort et lesquelles ont un copyleft faible ?

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. Décrivez quelques problèmes de compatibilité qui pourraient survenir lors de la combinaison d'un logiciel sous une licence copyleft faible avec un logiciel sous une licence non copyleft.

Alors qu'un copyleft fort exige que le logiciel modifié soit distribué sous la même licence, les conditions sont "assouplies" dans le cas d'un copyleft faible. Néanmoins, toute combinaison avec d'autres licences soulève des questions fondamentales de compatibilité. Des aspects importants doivent être pris en compte pour répondre à ces questions : Les parties originales des deux projets logiciels sont-elles clairement séparées dans la nouvelle œuvre et, si nécessaire, peuvent-elles encore faire l'objet d'une licence différente ? À quel niveau le lien entre les deux projets logiciels d'origine s'établit-il réellement ? Le code source est-il directement repris ou est-il "seulement" lié dynamiquement à l'autre logiciel (bibliothèque) ? L'une des deux licences permet-elle généralement le changement de licence ? Toutes ces

questions ne peuvent être résolues qu'après une analyse précise des licences respectives et avec l'aide d'un expert juridique spécialisé.



052.3 Licences de logiciels permissives

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 052.3

Valeur

3

Domaines de connaissance les plus importants

- Comprendre le concept des licences de logiciels permissives
- Comprendre les droits accordés par les licences de logiciels permissives
- Comprendre les obligations créées par les licences de logiciels permissives
- Comprendre les principales propriétés des licences logicielles permissives couramment utilisées
- Comprendre la compatibilité des licences logicielles permissives avec d'autres licences

Liste partielle de termes, fichiers et utilitaires utilisés

- Licence BSD à 2 clauses
- Licence BSD à 3 clauses
- Licence MIT
- Licence Apache, version 2.0



Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	052 Licences de logiciels à code source ouvert
Objectif :	052.3 Licences de logiciels permissives
Leçon :	1 sur 1

Introduction

Les licences *permissives* sont actuellement les licences de logiciels à code source ouvert les plus utilisées, contrairement aux licences *restrictives* telles que la licence publique générale GNU (GPL). Les licences logicielles permissives ont tendance à être simples et flexibles, et offrent un large éventail de libertés à leurs auteurs — peut-être la plus grande liberté disponible parmi les licences open source.

Ce type de licence accorde une grande liberté aux développeurs de logiciels en ce qui concerne l'utilisation, la modification et la redistribution du logiciel, à condition qu'ils mentionnent l'auteur original. Par exemple, quelqu'un peut généralement distribuer une œuvre dérivée sous une licence à code source fermé, à condition que l'œuvre comprenne l'attribution de l'auteur de l'œuvre originale dont la nouvelle est dérivée.

Le principe qui sous-tend cette logique est de permettre la plus grande diffusion possible du logiciel. Les licences permissives prétendent profiter aux individus et au public en facilitant l'utilisation commerciale du logiciel, tout en préservant les droits de l'auteur original par le biais de l'attribution.

Ce n'est pas une coïncidence si les premières et les plus importantes licences logicielles permissives ont été développées dans le domaine universitaire, comme les licences de type BSD de l'université de Berkeley en Californie ou la licence MIT/X11 de l'Institut de technologie du Massachusetts. Ces licences sont connues sous le nom de licences à code source ouvert académiques. Leur influence au sein de l'industrie a été telle qu'elles ont jeté les bases de licences logicielles permissives similaires conçues en dehors des contextes universitaires, telles que la licence Apache de l'Apache Software Foundation.

Droits et obligations des licences de logiciel permissives

En général, les licences logicielles permissives les plus courantes accordent au titulaire de la licence le droit illimité de :

Utiliser le logiciel

Les logiciels couverts par une licence permissive peuvent être utilisés par n'importe qui (des utilisateurs individuels aux sociétés commerciales, en passant par les autorités publiques) et à n'importe quelle fin, qu'elle soit personnelle ou professionnelle.

Modifier le logiciel

Le logiciel peut être amélioré, adapté ou même intégré en tant que composant (par exemple, une bibliothèque) dans d'autres logiciels.

Redistribuer le logiciel

Si le logiciel est modifié et prend la forme d'une œuvre dérivée, il peut être redistribué sous différentes licences, y compris des licences propriétaires.

La seule obligation contenue dans les licences de logiciel permissives est généralement celle de l'attribution : Le titulaire de la licence est tenu d'indiquer le nom de l'auteur original du logiciel dans l'œuvre dérivée et d'inclure une copie du texte de la licence dans toute redistribution du logiciel.

Caractéristiques des principales licences logicielles permissives

Cette section explique les différences entre la licence MIT/X11, les licences BSD les plus courantes et la licence Apache 2.0, qui sont toutes largement utilisées de nos jours.

Licence MIT/X11

La licence MIT/X11 (texte : <https://opensource.org/license/mit>), également connue sous le nom de licence X11, est l'une des licences académiques mentionnées précédemment. Elle tire son nom du

logiciel X Window System développé par le Massachusetts Institute of Technology en 1987. Cette licence est l'une des licences logicielles permissives les plus anciennes et les plus populaires, notamment en raison de son langage simple et clair.

Cette licence accorde à son détenteur les droits de :

- Utiliser, modifier et distribuer le logiciel
- Commercialiser le logiciel, avec ou sans modifications
- Publier des œuvres dérivées sous une licence différente, même sous forme de logiciel à code source fermé

La seule obligation du titulaire de la licence est d'inclure l'avis de droit d'auteur et le texte de la licence dans le code source du logiciel ou de ses œuvres dérivées.

La licence MIT/X11 est considérée comme compatible avec toutes les licences copyleft les plus importantes, qu'elles soient faibles ou fortes. En particulier, la licence MIT/X11 est compatible avec la licence :

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

Cela signifie que les œuvres dérivées d'un logiciel initialement sous licence MIT/X11 peuvent être redistribués sous l'une des licences susmentionnées ou être inclus dans des projets publiés sous l'une des licences susmentionnées.

Licence BSD à 2 clauses

La licence BSD à deux clauses (texte : <https://opensource.org/license/bsd-2-clause>) dérive de la licence BSD originale, créée en 1980 par l'université de Berkeley en Californie comme licence pour BSD, son système d'exploitation de type Unix.

Cette licence est réputée pour sa simplicité et, comme son nom l'indique, elle ne comporte que deux clauses courtes. Elle est pratiquement identique à la licence MIT/X11. Comme la licence MIT/X11, elle exige que le logiciel soit attribué. En outre, la licence BSD à deux clauses requiert du détenteur de la licence qu'il inclue le texte de la licence dans la documentation et les autres éléments fournis lors de la redistribution du code.

En résumé, la licence BSD à deux clauses accorde les droits suivants :

- Utiliser, modifier et distribuer le logiciel

- Commercialiser le logiciel, avec ou sans modification
- Publier des œuvres dérivées sous une licence différente, même sous forme de logiciel à code source fermé

Le titulaire de la licence a l'obligation de :

- Inclure l'avis de copyright et le texte de la licence dans le code source et le binaire du logiciel ou de ses œuvres dérivées
- Inclure l'avis de copyright et le texte de la licence dans la documentation ou tout autre matériel fourni avec le logiciel

La licence BSD à 2 clauses est considérée comme compatible avec toutes les principales licences copyleft, qu'elles soient faibles ou fortes. En particulier, la licence BSD à 2 clauses est compatible avec :

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

Par conséquent, les œuvres dérivées d'un logiciel initialement sous licence BSD à deux clauses peuvent être redistribués sous l'une des licences susmentionnées ou être inclus dans des projets publiés sous l'une des licences susmentionnées.

Licence BSD à 3 clauses

La licence BSD à 3 clauses (texte : <https://opensource.org/license/bsd-3-clause>) est une autre variante de la licence BSD originale, composée de trois clauses seulement. La principale caractéristique qui distingue cette licence de la licence BSD sœur à deux clauses est la “clause de non-approbation”, qui vise à empêcher que le nom de l'auteur original soit exploité pour distribuer ou vendre des œuvres dérivées.

La licence BSD à trois clauses accorde à son détenteur les droits de :

- Utiliser, modifier et distribuer le logiciel
- Commercialiser le logiciel, avec ou sans modifications
- Publier des œuvres dérivées sous une licence différente, même sous forme de logiciel à code source fermé

Le titulaire de la licence a l'obligation de :

- Inclure l’avis de copyright et le texte de la licence dans le code source et le binaire du logiciel ou de ses œuvres dérivées
- Inclure l’avis de copyright et le texte de la licence dans la documentation ou tout autre matériel fourni avec le logiciel
- S’abstenir de citer le nom du détenteur du copyright ou des contributeurs pour approuver ou promouvoir des produits dérivés de ce logiciel, sans autorisation écrite préalable spécifique

Pour expliquer cette dernière obligation — la “clause de non-approbation” — imaginons un scénario dans lequel une entreprise ou un particulier crée une œuvre dérivée à partir d’un logiciel sous licence permissive. En l’absence de cette clause, le titulaire de la licence pourrait promouvoir la nouvelle œuvre en indiquant qu’il s’agit d’un dérivé du logiciel d’un développeur bien connu, afin de profiter de son prestige.

Comme sa sœur, la licence BSD à trois clauses est considérée comme compatible avec toutes les licences copyleft les plus importantes, qu’elles soient faibles ou fortes. En particulier, la licence BSD à 3 clauses est compatible avec :

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

Par conséquent, les œuvres dérivées d’un logiciel initialement sous licence BSD à deux clauses peuvent être redistribués sous l’une des licences susmentionnées ou être inclus dans des projets publiés sous l’une des licences susmentionnées.

Licence Apache 2.0

La première licence Apache a été développée par l’Apache Software Foundation, une organisation à but non lucratif créée en 1999, pour distribuer ses logiciels et faciliter leur intégration dans d’autres projets. La mission était de garantir le développement collaboratif de logiciels, en adoptant la philosophie du logiciel open source avec une perspective favorable aux entreprises.

La licence Apache 2.0 (texte : <https://www.apache.org/licenses/LICENSE-2.0>), probablement la plus répandue parmi les licences logicielles permissives, diffère des licences examinées précédemment sur quelques points importants. Le principal aspect concerne l’attribution de droits de brevet à chaque détenteur de licence, y compris une clause relative à la résiliation de la licence de brevet, afin de limiter et d’empêcher toute demande de dommages-intérêts pour violation de brevet.

Deux autres obligations sont importantes. Le titulaire de la licence doit diffuser, sous la même licence Apache 2.0, toutes les parties ou tous les composants du logiciel qu’il n’a pas modifiés. En

outre, le titulaire de la licence doit placer des avis bien visibles dans tous les fichiers modifiés, indiquant que le titulaire de la licence a modifié ces fichiers.

La Free Software Foundation, qui promeut les licences restrictives, recommande Apache 2.0 comme la meilleure des licences permissives pour la distribution de petites quantités de logiciels et de bibliothèques.

La licence Apache 2.0 accorde au détenteur de la licence :

- Le droit d'utiliser, de modifier et de distribuer le logiciel
- Le droit de commercialiser le logiciel, avec ou sans modifications
- Le droit de publier des œuvres dérivées sous une licence différente, même en tant que logiciel à code source fermé
- Une licence de brevet pour utiliser, vendre, importer et transférer de toute autre manière le logiciel couvert par un brevet
- Une licence de brevet pour utiliser, vendre, importer et transférer de toute autre manière le logiciel qui pourrait autrement enfreindre la revendication de brevet d'un contributeur

Le titulaire de la licence a l'obligation de :

- Placer des avis bien visibles dans tous les fichiers modifiés, indiquant que les fichiers ont été modifiés
- Publier toutes les parties non modifiées du logiciel original sous la licence Apache 2.0
- Inclure l'avis de copyright et le texte de la licence dans le code source du logiciel et de ses œuvres dérivées
- Inclure l'avis de droit d'auteur et le texte de la licence dans la documentation et les autres éléments fournis avec le logiciel

La licence Apache 2.0 est considérée comme compatible avec certaines des principales licences copyleft. En particulier, la licence Apache 2.0 est compatible avec :

- GNU General Public License (GPL), version 3.0 mais pas la version 2.0
- GNU Lesser General Public License (LGPL), version 3.0 mais pas la version 2.0
- Mozilla Public License (MPL), version 2.0 mais pas la version 1.1

Cela signifie que les œuvres dérivées d'un logiciel initialement sous licence Apache 2.0 peuvent être redistribués sous l'une des licences compatibles susmentionnées ou être inclus dans des projets publiés sous l'une des licences compatibles susmentionnées.

La compatibilité limitée entre la licence Apache 2.0 et d'autres licences open source est principalement due à la présence de clauses relatives à l'octroi d'une licence de brevet au titulaire de la licence.

Licences de logiciels permissives par rapport aux autres licences open source

Maintenant que nous avons une vue d'ensemble des caractéristiques communes et essentielles des licences de logiciels permissives, nous pouvons examiner en quoi elles diffèrent des logiciels du domaine public et des licences copyleft.

Comparaison avec les logiciels du domaine public

La première distinction entre les licences de logiciels permissives et la mise à disposition de logiciels dans le domaine public réside dans leur existence même. Le domaine public n'est pas une licence à proprement parler, mais simplement un moyen de diffuser des logiciels. En d'autres termes, par définition, les œuvres du domaine public n'ont pas de licence.

La distinction suivante réside dans les obligations que les licences permissives imposent au titulaire de la licence. En effet, l'utilisateur d'une version du domaine public n'a aucune obligation. Cependant, toute personne qui utilise, modifie ou redistribue un logiciel sous licence permissive doit se conformer à l'obligation d'attribution expliquée plus haut : l'obligation d'inclure le nom de l'auteur original et une copie du texte de la licence dans le logiciel.

L'obligation d'attribution a pour conséquence qu'aucune œuvre dérivée provenant d'un logiciel sous licence permissive ne peut être placée dans le domaine public, car cela violerait (ou du moins contournerait) le droit de l'auteur original à recevoir l'attribution.

Comparaison avec le Copyleft

La distinction entre les licences de logiciels permissives et les licences copyleft restrictives est plus complexe, en particulier si l'on considère les différences entre les diverses licences copyleft. Comme nous l'avons vu dans les leçons précédentes, une différence majeure sépare les licences copyleft *fortes* (y compris la GNU General Public License (GPL), version 2.0 et 3.0) des licences copyleft *faibles* (y compris la GNU Lesser General Public License (LGPL), version 2.0 et 3.0, et la Mozilla Public License).

La principale différence entre les licences restrictives et les licences permissives réside dans le principe du copyleft, qui est au cœur des licences restrictives. En vertu de ce principe, le titulaire d'une licence qui modifie un logiciel sous licence copyleft, puis le distribue, doit nécessairement publier — totalement ou partiellement — l'œuvre dérivée sous la même licence que le logiciel

original. Le non-respect de cette obligation entraîne une violation du droit d'auteur, avec les conséquences juridiques qui en découlent.

Au contraire, les licences de logiciels permissives n'imposent pas une telle obligation. Ceux qui utilisent ou prévoient de publier des logiciels sous ce type de licence peuvent décider librement de la licence pour leur œuvre dérivée, y compris les licences propriétaires.

Copyleft fort

La différence entre les licences de logiciels permissives et les licences à copyleft fort est plus nette que dans le cas des licences à copyleft faible.

La différence essentielle est que les licences à copyleft fort exigent que les œuvres dérivées soient publiées sous la même licence que le logiciel original. Ceci s'applique également si le logiciel sous licence copyleft est intégré dans un autre projet : L'ensemble de l'œuvre dérivée doit être publié sous la même licence à copyleft fort.

Copyleft faible

Contrairement aux licences à copyleft fort, les licences à copyleft faible ne prévoient qu'une obligation partielle de publier une œuvre dérivée sous la même licence. Plus précisément, le titulaire de la licence qui redistribue un logiciel publié sous une licence à copyleft faible doit appliquer la même licence à la partie de son œuvre dérivée de l'œuvre originale. Par exemple, une bibliothèque basée sur une bibliothèque sous licence LGPL doit également être sous licence LGPL.

Les licences à copyleft faible ont été conçues précisément pour rapprocher leurs caractéristiques de celles des licences de logiciels permissives. Elles se distinguent néanmoins des licences de logiciels permissives en ce que ces dernières n'imposent en aucun cas l'obligation de maintenir la même licence sur une œuvre dérivée ou intégrée.

Exercices guidés

1. Quelles sont les deux obligations généralement imposées par les licences de logiciels permissives ?

--

2. Laquelle des licences suivantes n'est *pas* une licence de logiciels permissive ?

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

3. Qu'est-ce qui distingue une licence de logiciels permissive d'une licence copyleft ?

Le logiciel publié sous une licence copyleft ne peut pas être distribué, alors que le logiciel sous licence permissive peut l'être.	
Les œuvres dérivées d'un logiciel sous licence copyleft ne peuvent pas être diffusées sous une licence propriétaire, alors qu'un logiciel sous licence permissive peut l'être.	
Les licences copyleft ne sont légalement reconnues qu'aux États-Unis d'Amérique, alors que les licences logicielles permissives sont reconnues dans le monde entier.	

4. Quelle licence de logiciels permissive accorde une licence de brevet permettant d'utiliser, de vendre, d'importer et de transférer de toute autre manière le logiciel couvert par un brevet ?

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

5. Lorsqu'un logiciel est publié sous la licence MIT/X11, pouvez-vous le distribuer sous une licence propriétaire et vendre une œuvre dérivée basée sur ce logiciel ?

Exercices d'exploration

1. Vous modifiez un logiciel distribué sous la licence Apache 2.0 et redistribuez l'œuvre dérivée sous une licence propriétaire. Quelles sont les étapes à suivre pour respecter les obligations de la licence Apache 2.0 ?

2. Citez au moins trois exemples de projets populaires publiés sous des licences de logiciels permissives.

3. Pourquoi ne pouvez-vous pas distribuer, sous une licence LGPL 2.0, un logiciel qui comprend des composants publiés à l'origine sous la licence MIT/X11, la licence Apache 2.0 et la licence BSD à deux clauses ? Quelle autre licence à copyleft faible peut être utilisée pour distribuer le logiciel ?

Summary

Dans cette leçon, vous avez appris :

- Ce que sont les licences de logiciels permissives, les droits qu'elles accordent et les obligations qu'elles prévoient
- Les différences entre les licences de logiciels permissives et les autres licences open source
- Les caractéristiques des licences de logiciels permissives les plus populaires
- Compatibilité des licences de logiciels permissives avec les autres licences open source

Réponses aux exercices guidés

1. Quelles sont les deux obligations généralement imposées par les licences de logiciels permissives ?

L'obligation d'indiquer le nom de l'auteur original du logiciel et d'inclure une copie du texte de la licence dans l'œuvre dérivée.

2. Laquelle des licences suivantes n'est *pas* une licence de logiciels permissive ?

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

3. Qu'est-ce qui distingue une licence de logiciels permissive d'une licence copyleft ?

Le logiciel publié sous une licence copyleft ne peut pas être distribué, alors que le logiciel sous licence permissive peut l'être.	
Les œuvres dérivées d'un logiciel sous licence copyleft ne peuvent pas être diffusées sous une licence propriétaire, alors qu'un logiciel sous licence permissive peut l'être.	X
Les licences copyleft ne sont légalement reconnues qu'aux États-Unis d'Amérique, alors que les licences logicielles permissives sont reconnues dans le monde entier.	

4. Quelle licence de logiciels permissive accorde une licence de brevet permettant d'utiliser, de vendre, d'importer et de transférer de toute autre manière le logiciel couvert par un brevet ?

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

5. Lorsqu'un logiciel est publié sous la licence MIT/X11, pouvez-vous le distribuer sous une licence

propriétaire et vendre une œuvre dérivée basée sur ce logiciel ?

Oui.

Réponses aux exercices d'exploration

1. Vous modifiez un logiciel distribué sous la licence Apache 2.0 et redistribuez l'œuvre dérivée sous une licence propriétaire. Quelles sont les étapes à suivre pour respecter les obligations de la licence Apache 2.0 ?

Vous devez :

- Insérer dans chaque fichier modifié un avis attestant que les fichiers ont été modifiés.
 - Diffuser toutes les parties non modifiées du logiciel original sous la licence Apache 2.0.
 - Inclure l'avis de copyright et le texte de la licence dans le code source du logiciel ou de ses œuvres dérivées.
 - Inclure l'avis de copyright et le texte de la licence dans la documentation et les autres éléments fournis avec la distribution du logiciel.
2. Citez au moins trois exemples de projets populaires publiés sous des licences de logiciels permissives.
 - 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. Pourquoi ne pouvez-vous pas distribuer, sous une licence LGPL 2.0, un logiciel qui comprend des composants publiés à l'origine sous la licence MIT/X11, la licence Apache 2.0 et la licence BSD à deux clauses ? Quelle autre licence à copyleft faible peut être utilisée pour distribuer le logiciel ?

Même si la licence MIT/X11 et la licence BSD à deux clauses sont compatibles avec la licence LGPL 2.0, la licence Apache 2.0 ne l'est pas. Un logiciel qui comprend des composants publiés sous la licence MIT/X11, la licence Apache 2.0 et la licence BSD à deux clauses peut être publié sous la licence LGPL 3.0, car elle est compatible avec toutes ces licences de logiciels permissives.



**Linux
Professional
Institute**

Thème 053: Licences de contenu ouvert



053.1 Concepts des licences de contenu ouvert

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 053.1

Valeur

2

Domaines de connaissance les plus importants

- Comprendre les types de contenu ouvert
- Comprendre ce qui constitue un contenu soumis au droit d'auteur
- Comprendre les œuvres dérivées de matériel protégé par le droit d'auteur
- Comprendre la nécessité des licences de contenu ouvert
- Sensibilisation aux marques déposées

Liste partielle de termes, fichiers et utilitaires utilisés

- Documentation
- Images
- Œuvres d'art
- Cartes
- Musique
- Vidéos
- Conceptions et spécifications du matériel
- Bases de données
- Flux de données

- Fils de données



Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	053 Licences de contenu ouvert
Objectif :	053.1 Concepts des licences de contenu ouvert
Leçon :	1 sur 1

Introduction

Imaginez la situation suivante : Frank est un écrivain qui souhaite mettre certaines de ses histoires à la disposition de tout le monde sur l'internet, mais sous certaines conditions. Il veut que son travail soit gratuit. Il veut que tout le monde puisse utiliser ses histoires sans en demander l'autorisation, mais il ne veut pas que son travail soit utilisé à des fins commerciales. Enfin, il souhaite que les gens le créditent correctement si les histoires sont utilisées à quelque fin que ce soit. Il aimerait pouvoir le faire sans s'engager dans des procédures juridiques compliquées. Nous verrons plus tard pourquoi un écrivain peut vouloir faire ces choix.

Emma est une cinéaste qui enseigne à des étudiants comment produire des courts métrages. Entre autres choses, les étudiants ont besoin d'une histoire pour créer leurs œuvres. Emma connaît les histoires de Frank et pense qu'elles sont parfaites pour son cours.

Que peut faire Frank pour permettre à Emma d'utiliser ses histoires ? Cette situation et bien d'autres peuvent être résolues en utilisant un modèle de *licence de contenu ouvert* (*open content licensing*). De nos jours, des millions d'œuvres protégées par le droit d'auteur peuvent être réutilisées par n'importe qui sans le consentement explicite du détenteur du droit d'auteur ni le paiement d'un droit de licence, grâce à leur distribution sous des licences de contenu ouvert. Des

ressources telles que Wikipedia, Flickr, OpenStreetMap, Unsplash et Jamendo sont quelques exemples des nombreuses plateformes qui utilisent ce modèle de licence.

Selon une définition très large, le *contenu ouvert* désigne toute œuvre (y compris, par exemple, les films, la musique, les images, les textes, les bases de données, les ensembles de données, la documentation, les cartes et les conceptions matérielles) dont l'utilisation gratuite et la redistribution sont autorisées par la loi sur le droit d'auteur. Cela comprend également les œuvres protégées à l'origine dont la durée de protection a expiré et qui se trouvent donc dans le domaine public. Une définition un peu plus étroite présuppose qu'une œuvre a été explicitement placée sous une licence de contenu ouvert par son auteur, ce qui permet l'utilisation et la distribution de l'œuvre sans aucun paiement ou autorisation. Le modèle de contenu ouvert ne s'oppose donc pas à la loi sur le droit d'auteur, mais la complète.

Principes fondamentaux du droit d'auteur

Les licences de contenu ouvert étant basées sur le droit d'auteur, il est nécessaire d'apprendre certains aspects de la législation sur le droit d'auteur afin de comprendre pourquoi les licences sont nécessaires et ce qu'elles autorisent.

Le droit d'auteur fait partie d'une catégorie juridique connue sous le nom de *propriété intellectuelle*. Le droit d'auteur protège les œuvres originales en accordant à leurs créateurs des droits légaux exclusifs pour contrôler certaines utilisations de leurs œuvres par d'autres personnes et sociétés. En général, cela signifie que personne d'autre ne peut copier, distribuer, exécuter publiquement, adapter ou faire presque autre chose que regarder ou lire l'œuvre sans l'autorisation du détenteur du droit d'auteur.

Les principes fondamentaux que nous examinerons dans cette section comprennent ce qui est protégé par le droit d'auteur, ainsi que les personnes qui contrôlent les droits et peuvent accorder l'autorisation de réutiliser une œuvre protégée par le droit d'auteur, comme Emma souhaite le faire dans notre exemple.

Comme l'indique explicitement l'organisation mondiale de la propriété intellectuelle (*World Intellectual Property Organization*) :

Le droit d'auteur protège deux types de droits. Les droits patrimoniaux permettent aux titulaires de percevoir une compensation financière pour l'exploitation de leurs œuvres par des tiers. Le droit moral permet aux auteurs et aux créateurs de prendre certaines mesures afin de préserver et protéger le lien qui les rattache à leur œuvre. L'auteur ou le créateur peut être le titulaire des droits patrimoniaux ou ces droits peuvent être cédés à un ou plusieurs titulaires des droits d'auteur. De nombreux pays n'autorisent pas la cession des droits moraux. Le droit d'auteur ne protège que l'expression de faits ou d'idées. Il ne permet

pas au détenteur du droit d'auteur de posséder ou de contrôler exclusivement l'idée. Par exemple, une illustration peut être protégée par le droit d'auteur, mais pas l'idée qui en est à l'origine.

— Organisation mondiale de la propriété intellectuelle, Comprendre le droit d'auteur et les droits connexes (traduction en français sur wipo.int)

Les droits patrimoniaux protégés par le droit d'auteur durent longtemps, généralement des décennies après la mort du créateur. Les droits moraux n'expirent jamais.

Le droit d'auteur confère des droits sur les œuvres créatives, telles que les créations littéraires et artistiques, qui doivent répondre à un certain critère d'originalité. L'œuvre doit être une création de son créateur et ne doit pas être copiée à partir d'une autre œuvre. (Des débats sont en cours sur la part d'intelligence artificielle qui peut être utilisée dans une œuvre sans qu'elle soit considérée comme originale).

Le droit d'auteur ne protège que l'expression de faits ou d'idées. Il ne permet pas au détenteur du droit d'auteur de posséder ou de contrôler exclusivement l'idée. Par exemple, une illustration peut être protégée par le droit d'auteur, mais pas l'idée qui en est à l'origine.

Le droit d'auteur est automatique lorsqu'une œuvre est créée et fixée sous une forme tangible, par exemple une œuvre d'art numérique ou une chanson. Cela signifie que les droits sont accordés au créateur sans même que l'œuvre soit officiellement déposée.

Les personnes qui défendent les licences de contenu ouvertes veulent promouvoir une culture libre et le développement d'un bien commun numérique. Ils ont trouvé le système des droits d'auteur trop restrictif et rigide, tant pour les utilisateurs que pour les créateurs. En créant des licences standard faciles à utiliser, les défenseurs du contenu ouvert simplifient l'utilisation et la distribution des œuvres protégées par le droit d'auteur.

Que peut-on protéger par le droit d'auteur ?

Les lois sur le droit d'auteur varient d'un pays à l'autre. Toutefois, il existe des accords internationaux visant à normaliser les lois sur le droit d'auteur. Les œuvres littéraires et artistiques originales peuvent être protégées par le droit d'auteur : par exemple, les œuvres d'art, la musique, la photographie, le cinéma, la télévision, la littérature et la programmation. Les règles particulières permettant de déterminer ce qui peut être protégé par le droit d'auteur et le degré d'originalité d'une œuvre varient en fonction de la région.

Parfois, ces catégories peuvent être très générales et s'appliquent à des œuvres qui comportent à la fois des éléments créatifs et des éléments strictement fonctionnels : Par exemple, un court métrage est considéré comme une œuvre artistique. Toutefois, certaines parties de ce film

peuvent ne pas être protégées par le droit d'auteur si elles ne répondent pas au critère d'originalité.

Œuvres dérivées

Reprenons la situation hypothétique du début de la leçon : Frank a finalement décidé de publier ses histoires sous une licence de contenu ouvert afin qu'elles puissent être utilisées dans les conditions qu'il a choisies. Puisque les histoires étaient sous cette licence, Emma les a utilisées dans sa classe et a décidé de montrer certains des courts métrages de ses élèves lors d'un festival du film. Dans ce cas, le contenu créé par les étudiants peut être considéré comme une *œuvre dérivée*.

Une œuvre dérivée ou *adaptation* est une œuvre basée sur des œuvres existantes, telle qu'une traduction, un arrangement musical, une dramatisation, une fictionnalisation, une version cinématographique, un enregistrement sonore, une reproduction artistique ou toute autre forme dans laquelle l'œuvre originale est transformée ou adaptée. L'œuvre dérivée devient une seconde œuvre distincte, indépendante de la première par sa forme. La transformation, la modification ou l'adaptation de l'œuvre doit être significative pour être originale et donc protégée par le droit d'auteur.

Caractéristiques communes des licences de contenu ouvert

Chaque licence de contenu ouvert reconnaît le droit d'auteur et stipule que, sans licence de l'auteur, toute personne utilisant l'œuvre serait en violation du droit d'auteur. Par conséquent, ces licences s'inscrivent dans le cadre du système global des droits d'auteur au lieu d'essayer de le renverser.

Les licences de contenu ouvert garantissent également que l'auteur de l'œuvre soit dûment crédité. Si les destinataires de l'œuvre la distribuent à un tiers, ils doivent s'assurer que l'auteur original est reconnu et crédité. De même, lorsque les destinataires modifient l'œuvre, la version dérivée doit clairement mentionner l'auteur de l'œuvre originale et l'endroit où celle-ci peut être trouvée.

Contrairement à la plupart des licences de droit d'auteur, qui imposent des conditions restrictives à l'utilisation de l'œuvre, les licences de contenu ouvert permettent aux utilisateurs de jouir de certaines libertés en leur accordant des droits. Certains de ces droits sont communs à la quasi-totalité des licences de contenu ouvert, comme le droit de copier l'œuvre et le droit de la distribuer. Selon la licence, l'utilisateur peut également avoir le droit de modifier l'œuvre, de créer des œuvres dérivées, d'exécuter l'œuvre, d'afficher l'œuvre et de distribuer les œuvres dérivées.

Les licences de contenu ouvert peuvent contrôler les travaux dérivés. Ces licences incluent normalement le droit de créer une œuvre dérivée et de la distribuer sur n'importe quel support. Si une personne concède une licence sur un tableau sous une licence de contenu ouvert, le droit de créer un autre tableau à partir de ce tableau peut également être accordé. Ces autorisations sont accordées à la condition que d'autres puissent utiliser librement l'œuvre dérivée, tout comme l'œuvre originale.

Par conséquent, une licence de contenu ouvert garantit normalement que les œuvres dérivées sont concédées sous les termes et conditions de la même licence de contenu ouvert. Mais cette obligation ne s'applique pas lorsque l'œuvre est incluse dans une compilation. Par exemple, si une personne crée un album de chansons, dont l'une est soumise à une licence de contenu ouvert, toutes les chansons peuvent ne pas être soumises aux mêmes conditions de licence.

Un autre aspect important des licences de contenu ouvert est le contrôle de l'utilisation commerciale d'une œuvre. Les personnes peuvent concéder leurs œuvres sous une licence de contenu ouvert tout en limitant les droits à des fins non commerciales. Il est également possible d'accorder tous les droits, y compris le droit d'utiliser l'œuvre à des fins commerciales.

Les licences de contenu ouvert n'empêchent pas les gens de gagner de l'argent avec leur travail. Si une œuvre est soumise à une licence non commerciale, un éditeur ou une autre entité commerciale peut la publier en concluant au préalable un accord avec les détenteurs des droits d'auteur. En d'autres termes, même après avoir publié une œuvre sur une base non commerciale, les auteurs peuvent vendre les droits d'auteur à une entité à but lucratif, à condition qu'il n'y ait pas d'exclusion de l'utilisation continue et non commerciale.

Importance des licences de contenu ouvert

Quels sont les avantages d'un modèle de contenu ouvert et pourquoi les gens utiliseraient-ils une licence de contenu ouvert pour distribuer leur travail créatif plutôt que de s'en remettre au modèle traditionnel du droit d'auteur ?

Les licences de contenu ouvert permettent aux œuvres de circuler plus largement que si elles étaient restreintes par défaut. Par conséquent, les nouveaux artistes pourraient bénéficier de ces licences, car ils peuvent devenir plus populaires et être reconnus par un plus grand nombre de personnes. En renonçant à certains contrôles (et éventuellement aux revenus qui les accompagnent), les créateurs de contenu peuvent être invités à davantage de spectacles, obtenir davantage de parrainages, multiplier les collaborations, etc.

Les licences de contenu ouvert peuvent être pratiques à l'ère de l'internet, car les créateurs peuvent publier leurs œuvres sans dépendre d'une autre personne ou d'une autre institution. Par exemple, un photographe qui souhaite exposer ses photos peut les publier sur un site web

personnel. De cette manière, il peut se faire connaître d'un public plus large sans passer par une agence ou une galerie.

En choisissant la bonne licence, les titulaires de droits peuvent maximiser la distribution et garder le contrôle de la commercialisation de leurs œuvres. Si des personnes souhaitent utiliser une œuvre à des fins commerciales, le créateur du contenu peut conserver le droit d'accorder ou de refuser l'autorisation. Même si l'utilisation commerciale est interdite à d'autres, les titulaires de droits peuvent toujours utiliser leur œuvre à des fins commerciales.

Outre la possibilité d'une distribution beaucoup plus large d'une œuvre, les licences de contenu ouvert augmentent également la sécurité juridique pour les utilisateurs et réduisent considérablement les coûts des transactions juridiques.

Les gens peuvent également utiliser des modèles de financement qui ne dépendent pas de l'utilisation d'une licence non commerciale. Par exemple, de nombreux artistes et créateurs utilisent le financement participatif (*crowdfunding*) pour financer leur œuvre avant de la publier sous une licence permissive. D'autres utilisent un modèle dans lequel le contenu de base est gratuit, mais les suppléments tels que les versions imprimées ou l'accès spécial à un site web réservé aux membres sont pour les clients payants uniquement.

Les licences de contenu ouvert permettent aux personnes de distribuer leurs œuvres à n'importe qui et sur n'importe quel support et format, tels que les sites web, les photocopies, les CD ou les livres, sans aucune restriction.

Les licences de contenu ouvert établissent automatiquement une licence entre les auteurs et les utilisateurs. Sans une licence de contenu ouvert, le partage d'œuvres via une autre source en ligne nécessiterait un accord contractuel individuel entre les auteurs et les utilisateurs.

Marques déposées et droits d'auteur

Les marques déposées, comme les droits d'auteur, sont un type de propriété intellectuelle. Mais la loi qui protège une marque déposée est différente de celle qui protège le droit d'auteur. Les marques déposées protègent les marques, les mots apparentés tels que les noms de marque, les logos, les symboles et même les sons et les couleurs qui sont utilisés pour distinguer des produits et des services particuliers d'autres produits et services. Tout élément spécial utilisé pour promouvoir et distinguer les entreprises et les services ou produits vendus des autres peut être une marque déposée.

Le détenteur d'une marque déposée est généralement autorisé à empêcher d'autres personnes d'utiliser ces articles protégés par la marque déposée si le public risque d'être confondu. Le droit des marques déposées aide les producteurs de biens et de services à protéger leur réputation et

protège le public en lui donnant un moyen simple de différencier des produits et des services similaires.

Les marques déposées peuvent être enregistrées officiellement ou bénéficier d'une protection automatique en vertu du droit coutumier. Cela signifie qu'une marque déposée existe dès qu'elle est utilisée, mais une marque déposée en vertu du droit commun n'offre pas la même protection juridique qu'une marque déposée enregistrée. C'est pourquoi de nombreuses entreprises enregistrent leurs marques déposées.

Le droit d'auteur et les marques déposées peuvent coexister. Par exemple, Wikipédia est une marque déposée et son logo est également protégé par le droit d'auteur en tant qu'œuvre d'art ou création originale.

Exercices guidés

1. Les licences de contenu ouvert peuvent-elles être utilisées pour contourner le droit d'auteur ?

2. Qu'est-ce qui est considéré comme du contenu ouvert ?

3. Qu'est-ce qu'une œuvre dérivée ?

Exercices d'exploration

1. Que feriez-vous si vous souhaitiez publier une œuvre et autoriser les gens à l'utiliser à toutes fins, à condition qu'ils redistribuent l'œuvre en respectant les mêmes droits et conditions ?

2. Pouvez-vous distribuer des œuvres dérivées commerciales basées sur des œuvres publiées sous une licence de contenu ouvert ?

Résumé

Dans cette leçon, vous avez appris :

- Principes fondamentaux de la législation sur le droit d'auteur
- Ce qui peut être considéré comme un contenu protégé par le droit d'auteur
- Ce qu'est une œuvre dérivée ou une adaptation
- Caractéristiques communes aux licences de contenu ouvert
- Types de contenu ouvert
- Importance et avantages du modèle de licence de contenu ouvert
- Le droit d'auteur et les marques déposées en tant que types de propriété intellectuelle

Réponses aux exercices guidés

1. Les licences de contenu ouvert peuvent-elles être utilisées pour contourner le droit d'auteur ?

Les licences de contenu ouvert ne peuvent pas être utilisées pour contourner le droit d'auteur. Les licences de contenu ouvert sont un type de licence de droit d'auteur qui accorde certains droits sous certaines conditions, tout en maintenant les droits exclusifs du détenteur du droit d'auteur.

2. Qu'est-ce qui est considéré comme du contenu ouvert ?

Le contenu ouvert peut être toute œuvre protégée par le droit d'auteur et publiée sous une licence de contenu ouvert. Il peut s'agir de films, de musique, d'images, de textes, de bases de données, de documentation, de cartes, de conceptions matérielles et d'autres créations. Ces types de licences accordent des droits tels que l'utilisation, la redistribution, la création d'œuvres dérivées et l'utilisation commerciale ou non commerciale sans autorisation spéciale.

3. Qu'est-ce qu'une œuvre dérivée ?

Une œuvre dérivée, également appelée adaptation, est une œuvre basée sur une œuvre existante dans laquelle l'œuvre originale est transformée ou adaptée. Les traductions, les arrangements musicaux, les dramatisations, les versions cinématographiques, les enregistrements sonores et les reproductions artistiques sont des exemples d'œuvres dérivées.

Réponses aux exercices d'exploration

1. Que feriez-vous si vous souhaitiez publier une œuvre et autoriser les gens à l'utiliser à toutes fins, à condition qu'ils redistribuent l'œuvre en respectant les mêmes droits et conditions ?

Vous pouvez mettre l'œuvre à disposition sous une licence de type copyleft. De cette manière, toutes les œuvres dérivées de l'œuvre originale doivent également être ouvertes, en exigeant qu'elles soient publiées sous la même licence ou toute autre licence compatible.

2. Pouvez-vous distribuer des œuvres dérivées commerciales basées sur des œuvres publiées sous une licence de contenu ouvert ?

Cela dépend de la licence sous laquelle l'œuvre originale est disponible. Si la licence stipule que vous pouvez utiliser les œuvres dérivées à n'importe quelle fin, même commerciale, alors vous pouvez le faire.



053.2 Licence Creative Commons

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 053.2

Valeur

2

Domaines de connaissance les plus importants

- Comprendre le concept des licences Creative Commons
- Comprendre les types de licences Creative Commons et leurs combinaisons
- Comprendre les droits accordés par les licences Creative Commons
- Comprendre les obligations créées par les licences Creative Commons

Liste partielle de termes, fichiers et utilitaires utilisés

- Attribution du domaine public (CC0)
- Creative Commons Attribution (CC BY)
- Creative Commons Attribution-Partage dans les mêmes conditions (CC BY-SA)
- Creative Commons Attribution-Pas d'utilisation commerciale (CC BY-NC)
- Creative Commons Attribution-Pas d'utilisation commerciale-Partage dans les mêmes conditions (CC BY-NC-SA)
- Creative Commons Attribution-Pas de modification (CC BY-ND)
- Creative Commons Attribution-Pas d'utilisation commerciale-Pas de modification (CC BY-NC-ND)



Linux
Professional
Institute

Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	053 Licences de contenu ouvert
Objectif :	053.2 Licences Creative Commons
Leçon :	1 sur 1

Introduction

Le succès des logiciels libres depuis les années 1990, ainsi que le triomphe simultané de l'internet, ont éveillé dans d'autres secteurs le désir de bénéficier de conditions similaires pour soutenir le développement et la distribution d'œuvres créatives, c'est-à-dire d'œuvres qui sont fondamentalement soumises au droit d'auteur. Ces souhaits reflètent une variété d'intérêts.

Les artistes créateurs veulent déterminer eux-mêmes les conditions dans lesquelles leurs œuvres peuvent être utilisées par d'autres. Dans le même temps, ils ont également intérêt à utiliser les œuvres d'autrui pour eux-mêmes, comme cela s'est toujours fait dans le cadre des processus créatifs, par exemple en les citant, en les éditant ou en les adaptant. Pour les destinataires des œuvres, des questions se posent quant aux possibilités d'utilisation, par exemple si et sous quelle forme une œuvre peut être reproduite ou transmise.

En plus de clarifier ces questions pratiques, les règles doivent être élaborées de manière à ce qu'elles soient conformes aux normes juridiques applicables — principalement la loi sur le droit d'auteur — ce qui est rendu plus difficile par les diverses façons dont la loi sur le droit d'auteur est réglementée à l'échelle internationale.

Enfin, les règles doivent être faciles à comprendre et à appliquer, afin d'accélérer les processus de création et de donner aux créateurs et aux destinataires une sécurité juridique.

Origine et objectifs de Creative Commons

Les premières licences dans le domaine des logiciels libres, en particulier la licence publique générale GNU, ont fait œuvre de pionnier, car elles ont réussi à fournir un cadre juridique fiable pour les processus entièrement nouveaux entourant le développement collaboratif de logiciels.

En 2001, un groupe dirigé par le professeur de droit Lawrence Lessig de la Stanford Law School a fondé une organisation à but non lucratif appelée *Creative Commons* (CC), inspirée des logiciels libres. L'objectif de l'organisation est résumé sur le site web du projet comme suit :

Creative Commons est une organisation à but non lucratif qui permet le partage et la réutilisation de la créativité et des connaissances grâce à la fourniture d'outils juridiques gratuits.

— Creative Commons, Site web (FAQ)

Avec le terme « communs », qui fait référence à une forme d'activité économique communautaire déjà documentée au Moyen Âge, l'organisation met l'accent sur un besoin historiquement vérifiable et profondément humain d'action collective orientée vers le bien commun. La tâche qu'elle s'est imposée est donc de créer un cadre juridique moderne pour le travail créatif. Creative Commons transpose les exigences et les solutions du mouvement du logiciel libre à d'autres domaines créatifs tels que la littérature, les arts du spectacle (peinture, graphisme, photographie, vidéo, etc.), la musique, mais aussi la documentation et les travaux scientifiques, c'est-à-dire, en gros, toutes les œuvres soumises au droit d'auteur.

À l'instar de la Free Software Foundation, Creative Commons choisit également de mettre en œuvre ses objectifs par le biais de *licences* : des compilations de spécifications normalisées et juridiquement contraignantes que les créateurs attribuent explicitement à leurs œuvres, généralement avant de les “diffuser” au public, c'est-à-dire de les publier.

Le principe “tous droits réservés” inscrit dans la loi américaine sur les droits d'auteur est perçu comme trop restrictif au regard des possibilités techniques en constante évolution qui s'offrent aux processus créatifs. Les créateurs eux-mêmes ne savent souvent pas très bien dans quelle mesure ils sont autorisés à exploiter le travail d'autrui sans dépasser les limites de la loi sur le droit d'auteur et, dans le pire des cas, sans s'exposer à des poursuites judiciaires. Creative Commons oppose à ce régime autoritaire le principe de “certains droits réservés” : Les créateurs désignent les droits qu'ils réservent et renoncent ainsi à tous les autres.

La combinaison de quatre conditions de base simples (modules) permet d'obtenir un total de six

licences, parmi lesquelles les auteurs peuvent choisir et attribuer celle qui convient à leur travail.

Modules de la licence Creative Commons

Les quatre modules que nous venons de mentionner correspondent à quatre décisions de base simples qu'un auteur prend en ce qui concerne l'utilisation et la distribution d'une œuvre. Chaque module peut être représenté par une abréviation de deux lettres et un symbole. La définition respective est très courte et compréhensible même pour les profanes en droit, ce qui contribue grandement à la popularité des licences CC. Dans certains cas, cependant, les clauses peuvent conduire à des questions non résolues ou à des zones d'ombre.

Attribution (BY)

Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que l'offrant vous soutient ou soutient la façon dont vous avez utilisé son œuvre.

— Creative Commons, Attribution

Le mot anglais “_by_” indique que les auteurs doivent être nommés, c'est-à-dire que l'œuvre doit être clairement attribuée aux auteurs si elle est utilisée, reproduite ou transmise sous quelque forme que ce soit. Le module BY est le seul qui soit obligatoire dans *toutes* les licences CC. En d'autres termes, aucune œuvre sous les six licences CC standard ne peut échapper à l'attribution.

Cette simple exigence peut poser des problèmes pratiques dans les projets collaboratifs développés sur l'internet. Par exemple, dans les travaux dérivés de l'encyclopédie en ligne Wikipedia, on peut se demander comment “l'attribution” peut être effectuée “de manière appropriée” avec des milliers de contributeurs.

Utilisation non commerciale (NC)

Vous n'êtes pas autorisé à faire un usage commercial de cette œuvre, tout ou partie du matériel la composant.

— Creative Commons, Utilisation non commerciale

L'intention du module d'*utilisation non commerciale* (*NonCommercial*) semble claire à première vue : Il vise à empêcher l'appropriation d'une œuvre librement disponible par d'autres à des fins commerciales. Il s'agit souvent d'œuvres qui ont été développées avec des fonds publics, par exemple dans les universités. Elles sont censées être “protégées” de la commercialisation afin de garantir leur qualité et leur disponibilité gratuite à long terme.

En fait, le module NC est souvent source d'incertitude dans la pratique, car il n'est pas du tout évident, au cas par cas, de déterminer si une utilisation est réellement commerciale. Les enseignants qui utilisent du matériel sous licence CC avec un module NC, par exemple, se trouvent déjà dans une zone d'ombre juridique s'ils enseignent dans une école privée ou une université privée où les étudiants doivent payer pour assister aux cours.

De nombreux critiques du module NC rejettent également l'argument selon lequel le matériel libre devient "non libre" à la suite d'une utilisation commerciale, car les œuvres sont toujours *également* disponibles gratuitement.

Pas d'œuvre dérivée (ND)

Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant l'œuvre originale, vous n'êtes pas autorisé à distribuer ou mettre à disposition œuvre modifiée.

— Creative Commons, Pas d'œuvre dérivée

Les *dérivés* (*Derivatives* en abrégé : *Derivs*), c'est-à-dire les œuvres dérivées, ont déjà été mentionnés dans d'autres leçons. Dans le contexte des Creative Commons, le terme désigne également la création de nouvelles œuvres sur la base d'œuvres existantes couvertes par le droit d'auteur ; il peut s'agir de matériel pédagogique qu'un enseignant adapte pour ses cours, par exemple, ou de la traduction d'un roman dans une autre langue. *pas d'œuvre dérivée* (*NoDerivs*) exclut catégoriquement la distribution de modifications ou d'adaptations de ce type : L'œuvre ne peut être transmise que sous la forme publiée par l'auteur.

Partage dans les mêmes conditions (SA)

Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant l'œuvre originale, vous devez diffuser l'œuvre modifiée dans les mêmes conditions, c'est à dire avec la même licence avec laquelle l'œuvre originale a été diffusée.

— Creative Commons, Partage dans les mêmes conditions

Le module *partage dans les mêmes conditions* (*ShareAlike*) indique qu'une œuvre ne peut être partagée que dans les mêmes conditions que celles de la licence attribuée. Le module *ShareAlike* est considéré comme la contrepartie du principe du copyleft dans les licences de logiciels libres, qui garantit que les libertés accordées par la licence s'appliquent également de manière inchangée aux œuvres dérivées.

Le module SA est particulièrement intéressant en combinaison avec d'autres modules, car l'exigence SA met également à jour d'autres conditions définies, comme nous le verrons en

discutant des différentes licences.

Licences Creative Commons de base

Les combinaisons des modules que nous venons de présenter aboutissent à un total de six licences. Il n'y en a que six parce que toutes les combinaisons possibles des quatre modules n'ont pas de sens : Par exemple, l'obligation de partager des œuvres même modifiées dans les mêmes conditions (ShareAlike) et l'interdiction des adaptations (NoDerivs) s'excluent mutuellement. Par conséquent, il n'existe pas de licence qui contienne ces deux modules. Étant donné que l'attribution de l'auteur est également une composante de toutes les licences, il en résulte ce que l'on appelle les *licences de base*, dont nous parlerons plus loin.

La liste de ces licences peut également être imaginée comme une échelle allant de “plusieurs libertés” à “peu de libertés”, puisque l'auteur répond aux questions sur les possibilités d'utilisation ou leurs restrictions par rapport à l'œuvre, étape par étape, et reçoit finalement la licence souhaitée.

Creative Commons Attribution (CC BY)

L'attribution de l'auteur a déjà été mentionnée comme un élément obligatoire dans toutes les licences, et la première licence, *CC Attribution*, ne comprend donc que ce module. Une œuvre est donc autorisée à toute forme d'utilisation, de modification et de distribution tant que l'exigence d'attribution est respectée.



Figure 5. Icône CC BY

Par exemple, si un auteur a publié un poème sous CC BY, un éditeur pourrait l'inclure dans un recueil de poèmes sans consulter l'auteur et sans aucune obligation financière et le distribuer en librairie, par exemple, à condition que ce poème soit clairement identifié comme étant l'œuvre de l'auteur.

Creative Commons Attribution - Partage dans les mêmes conditions (CC BY-SA)

Attribution - Partage dans les mêmes conditions (CC Attribution-ShareAlike) correspond au CC BY, mais le complète par les exigences de *Partage dans les mêmes conditions*, c'est-à-dire la distribution de versions modifiées de l'œuvre dans les mêmes conditions.



Figure 6. Icône CC BY-SA

Par exemple, si une chanteuse place sa chanson sous la licence CC BY-SA, un autre groupe peut reprendre ou adapter cette œuvre, à condition de publier à son tour sa version de la chanson sous CC BY-SA ou une autre licence compatible avec celle-ci.

Creative Commons Attribution - Utilisation non commerciale (CC BY-NC)

La *CC Attribution - Utilisation non commerciale (CC Attribution-NonCommercial)* est la première licence de la série qui lie l'utilisation d'une œuvre à une prohibition en excluant la distribution et l'adaptation à des fins commerciales.

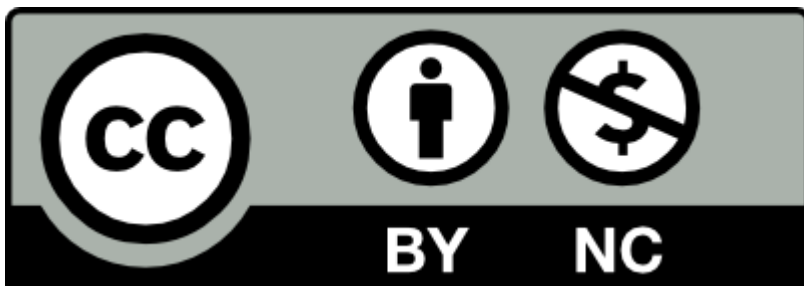


Figure 7. Icône CC BY-NC

Dans notre exemple, un groupe serait autorisé à adapter la chanson de la chanteuse, mais à ne pas utiliser cette adaptation à des fins commerciales, par exemple en ne la vendant pas sur ses propres disques ou en ne la jouant pas lors de concerts pour lesquels il perçoit un cachet. Toutefois, le groupe serait autorisé à interdire en plus les adaptations de sa version, c'est-à-dire à publier l'adaptation sous la licence CC BY-NC-ND.

Creative Commons Attribution - Utilisation non commerciale - Partage dans les mêmes conditions (CC BY-NC-SA)

Dans le cas de la *CC Attribution - Utilisation non commerciale - Partage dans les mêmes conditions* (*CC Attribution-NonCommercial-ShareAlike*), l'interdiction d'utilisation commerciale est liée à la distribution dans les mêmes conditions. Reprenons notre exemple : Bien que le groupe soit autorisé à reprendre la chanson de la chanteuse, il n'est pas autorisé à ajouter le module ND à la version reprise, car l'arrangement doit être distribué dans les mêmes conditions.



Figure 8. Icône CC BY-NC-SA

Creative Commons Attribution - Pas d'œuvre dérivée (CC BY-ND)

Tout comme l'utilisation non commerciale, la *CC Attribution - Pas d'œuvre dérivée* (*CC Attribution-NoDerivs*) est également basé sur une prohibition—dans ce cas, la prohibition de modifier ou d'adapter une œuvre. Le groupe de notre exemple ne serait donc pas autorisé à distribuer une reprise de la chanson de la chanteuse.

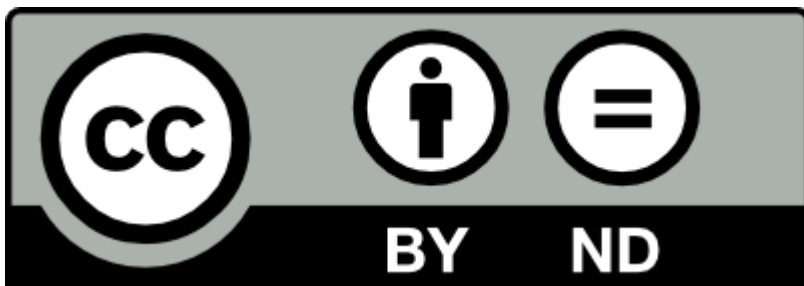


Figure 9. Icône CC BY-ND

L'utilisation de licences CC avec des clauses ND (c'est-à-dire la licence CC BY-ND et la licence CC BY-NC-ND présentée ci-dessous) est considérée avec scepticisme, en particulier dans le domaine scientifique, car elles peuvent entraver l'échange de connaissances nécessaire au progrès scientifique. Même sur le site de Creative Commons, les licences avec un module ND sont décrites comme incompatibles avec le principe de libre accès qui est populaire dans le monde scientifique. Comme exemple de la tendance trop restrictive de la ND, il est cité que même les traductions d'articles scientifiques sont considérées comme des œuvres dérivées et sont donc interdites.

Creative Commons Attribution - Utilisation non commerciale - Pas d'œuvre dérivée (CC BY-NC-ND)

La licence CC la plus restrictive, *CC Attribution - Utilisation non commerciale - Pas d'œuvre dérivée* (*CC Attribution-NonCommercial-NoDerivs*), combine la prohibition de l'utilisation commerciale avec celle de la distribution d'œuvres dérivées. En termes positifs, cela signifie qu'une œuvre ne peut être reproduite et distribuée que sous la forme publiée par l'auteur et qu'elle ne peut être utilisée qu'à des fins non commerciales.



Figure 10. Icône CC BY-NC-ND

La chanteuse de notre exemple empêche ainsi un groupe de reprendre sa chanson (ND), mais risque en même temps que sa chanson soit exclue de toute plateforme générant des revenus provenant de clients ou de la publicité.

Creative Commons Zéro (CC0) et la marque du domaine public

Il a déjà été mentionné que le droit d'auteur est réglementé de manière très différente selon les pays et les juridictions. Aux États-Unis, par exemple, un détenteur de droits peut renoncer totalement à ses droits d'auteur. Ce faisant, il transfère son œuvre dans ce que l'on appelle le *domaine public*, c'est-à-dire dans le cadre d'une utilisation générale et illimitée.

Dans la plupart des pays européens, toutefois, outre les droits d'utilisation purs, le droit d'auteur comprend également des droits personnels, qui ne sont généralement pas transférables et auxquels l'auteur ne peut donc pas renoncer. Si un auteur souhaite mettre son œuvre à la disposition du public, il renonce à tous les droits d'utilisation, mais reste l'auteur.

En outre, il existe des œuvres dont l'utilisation n'est soumise à aucune restriction en soi. Il s'agit, par exemple, d'œuvres dont la durée légale de protection a expiré (comme un roman 70 ans après la mort de son auteur), ou d'œuvres qui n'ont jamais été protégées en principe (comme les textes juridiques).

Creative Commons propose deux *outils du domaine public* pour identifier les œuvres qui sont à la disposition du grand public sans restriction.

La *Creative Commons Zéro* (CC0) n'impose pratiquement aucune condition à la reproduction, à la distribution et à la modification d'une œuvre. L'attribution obligatoire de l'auteur dans toutes les licences CC est également supprimée. Cette licence permet aux détenteurs de droits de marquer leurs œuvres comme faisant partie du domaine public dans le plus grand nombre possible de systèmes juridiques.



Figure 11. Icône CC0

Prenons l'exemple d'un document pédagogique élaboré à des fins d'enseignement. L'auteur ou les auteurs peuvent utiliser le label CC0 pour garantir que ce matériel peut être utilisé par tout le monde sans aucune restriction : Il peut être copié en tout ou en partie, distribué gratuitement ou contre rémunération, et intégré en tout ou en partie dans d'autres documents. Les auteurs originaux ne seront nommés nulle part.

Pour identifier les œuvres qui ne sont pas soumises à des restrictions de droit d'auteur, Creative Commons recommande l'utilisation de la *Marque du domaine public* :

La marque du domaine public fonctionne comme une étiquette ou un label, permettant à des institutions telles que celles-ci, ainsi qu'à d'autres personnes possédant ces connaissances, de communiquer qu'une œuvre n'est plus restreinte par le droit d'auteur et peut être utilisée librement par d'autres personnes.

— Creative Commons, Marque du domaine public (Traduction non officielle)



Figure 12. Icône marque du domaine public

Sélection de licences et marquage des œuvres

Il a déjà été mentionné que Creative Commons vise à être aussi simple que possible pour les donneurs de licence (créateurs de contenu) et les preneurs de licence (destinataires des œuvres). Concrètement, l'organisation offre une grande assistance depuis la sélection de la licence appropriée jusqu'à l'étiquetage de l'œuvre et son utilisation.

Le *sélecteur de licence CC* (*CC License Chooser*) sur le site web de l'organisation conduit pas à pas à la recommandation de la licence appropriée en posant des questions simples auxquelles l'auteur répond en fonction de l'utilisation souhaitée de son travail ([Sélecteur de licence CC](#)).

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
☒ I own or have authority to license the work.
☒ I have read and understand the terms of the license.
☒ 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. Sélecteur de licence CC

Si la licence proposée correspond aux intentions de l'auteur, il peut marquer son œuvre en conséquence. Le sélecteur de licence fournit un code HTML, par exemple, que l'auteur peut ajouter directement à un site web affichant l'œuvre ([Code HTML avec un lien vers la licence](#)).

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

```
vertical-align: text-bottom;" src="https://mirrors.creativecommons.org/presskit/icons/cc.svg?ref=chooser-v1"></a></p>
```

license abbreviation ☒ full license name

Copy

Figure 14. Code HTML avec un lien vers la licence

Le résultat est alors une notice standardisée avec le lien vers la licence correspondante ([Notice de licence avec lien vers la licence](#)).

This work is licensed under [CC BY-ND 4.0](#)   

Figure 15. Notice de licence avec lien vers la licence

Les icônes spécifiques aux licences, dans lesquelles les modules inclus sont directement visibles, ont déjà été introduites. Elles se sont imposées comme des “attrape-œil” permettant d’identifier les contenus libres sur l’internet.

Licences internationales et licences adaptées

Dès le départ, Creative Commons a mis l'accent sur le développement de licences aussi générales que possible, c'est-à-dire valables dans le monde entier, qu'elle identifie en conséquence par l'ajout de la mention "internationale" (anciennement aussi "non-adaptée" ou "unported"). D'autre part, il existe des variantes qui tiennent compte des particularités d'un système juridique régional ou national et qui sont désignées comme "adaptées" ("ported"). Par exemple, en plus de la licence *CC BY-SA 3.0 Unported*, il existe une version spéciale pour l'Allemagne sous le nom de *CC BY-SA 3.0 Germany*.

Avec la version actuelle 4.0 des licences CC, Creative Commons s'efforce de poursuivre la standardisation et a (jusqu'à présent) complètement renoncé aux versions portées. Dans la version 4.0, la licence "internationale" est expressément recommandée :

Nous vous recommandons d'utiliser une licence internationale version 4.0. Ceci est la version la plus mise à jour de nos licences, rédigée suite à une large consultation avec notre réseau mondial de sociétés affiliées, et a été écrite pour être valable à l'échelle internationale. Il n'y a pas de ports de 4.0, et il est prévu que quelques-unes seront créées.

— Creative Commons, FAQ

Exercices guidés

1. Quel module du système de licence Creative Commons fait partie des six licences Creative Commons ?

2. Quel module du système de licence Creative Commons exige la distribution d'une œuvre dans les mêmes conditions ?

3. Quelle est la différence entre CC0 et les six licences de base CC ?

4. Quelle est la différence entre les licences "internationales" et les licences "adaptées" ?

Exercices d'exploration

1. Est-il possible de publier une photographie d'une œuvre qui est dans le domaine public sous une licence CC ?

2. Un auteur peut-il publier son roman, dans lequel il incorpore un sonnet de Shakespeare dans son intégralité, sous une licence CC ?

3. Une utilisatrice publie une photo d'elle sur son site web sous licence CC. Peut-elle empêcher la diffusion de cette image en invoquant ses droits personnels ?

Résumé

Fondée en 2001, l'organisation Creative Commons a pour objectif de soutenir le partage et l'utilisation des œuvres créatives, c'est-à-dire des œuvres soumises au droit d'auteur, à l'aide d'outils juridiques gratuits. L'accent est mis sur quatre modules (Attribution, Utilisation non commerciale *NonCommercial*, Pas d'œuvre dérivée *NoDerivs* et Partage dans les mêmes conditions *ShareAlike*), qui sont combinés en six licences dites de base et peuvent être sélectionnées par les auteurs pour leurs œuvres.

Réponses aux exercices guidés

1. Quel module du système de licence Creative Commons fait partie des six licences Creative Commons ?

Le module “Attribution”, abrégé par “BY”.

2. Quel module du système de licence Creative Commons exige la distribution d’une œuvre dans les mêmes conditions ?

Le module “Partage dans les mêmes conditions” (“Share Alike”), abrégé par “SA”.

3. Quelle est la différence entre CC0 et les six licences de base CC ?

Avec CC0, l’auteur n’obtient pas certains droits — comme avec les autres licences CC — mais renonce à *tous* les droits sur l’œuvre dans le cadre des possibilités de la juridiction concernée. Ce principe de “pas de droits réservés” correspond à la désignation de l’œuvre comme *domaine public*.

4. Quelle est la différence entre les licences “internationales” et les licences “adaptées” ?

Jusqu’à la version 3.0, Creative Commons proposait des variantes de ses licences qui tenaient compte des spécificités d’une juridiction régionale ou nationale. Depuis la version 4.0, CC a renoncé à ces versions dites “adaptées” et recommande la version “internationale” de la licence concernée, valable dans le monde entier.

Réponses aux exercices d'exploration

1. Est-il possible de publier une photographie d'une œuvre qui est dans le domaine public sous une licence CC ?

Cela dépend si la photographie remplit les critères selon lesquels elle est elle-même protégée par le droit d'auteur. En d'autres termes, la photographie elle-même doit être reconnaissable en tant qu'œuvre créative et digne de protection, par exemple par la perspective, l'arrière-plan, les filtres, etc. Si c'est le cas, elle peut être publiée sous une licence CC et les conditions de la licence CC s'appliquent alors. Toutefois, la partie originale, c'est-à-dire le motif proprement dit, reste dans le domaine public.

2. Un auteur peut-il publier son roman, dans lequel il incorpore un sonnet de Shakespeare dans son intégralité, sous une licence CC ?

Oui, mais seules les parties de la création de l'auteur du roman sont soumises à la licence CC choisie par l'auteur. Le sonnet, qui est dans le domaine public, reste dans le domaine public.

3. Une utilisatrice publie une photo d'elle sur son site web sous licence CC. Peut-elle empêcher la diffusion de cette image en invoquant ses droits personnels ?

En publiant une photo sous licence CC, le propriétaire accepte généralement qu'elle soit diffusée. Une limite est atteinte lorsque l'utilisation de la photo viole les droits personnels du propriétaire, par exemple lorsque l'image est grossièrement déformée ou utilisée à des fins publicitaires ou dans un contexte politique sans le consentement du propriétaire. Les droits personnels du propriétaire ne sont pas affectés par la réglementation des droits d'utilisation par la licence CC, de sorte que dans de tels cas, il peut intenter une action en justice contre l'utilisation de sa photo.



053.3 Autres licences de contenu ouvert

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 053.3

Valeur

1

Domaines de connaissance les plus importants

- Comprendre les licences pour la documentation
- Comprendre les licences pour les ensembles de données et les bases de données
- Comprendre les droits accordés par les licences de contenu ouvert
- Comprendre les obligations créées par les licences de contenu ouvert

Liste partielle de termes, fichiers et utilitaires utilisés

- Licence de documentation libre GNU, version 1.3 (GFDL)
- Licence Open Data Commons Open Database (ODbL)
- Accord de licence des données communautaires – Permissive, version 1.0 (CDLA)
- Accord de licence des données communautaires – Partage, version 1.0 (CDLA)
- Accès ouvert



Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	053 Licences de contenu ouvert
Objectif :	053.3 Autres licences de contenu ouvert
Leçon :	1 sur 1

Introduction

Les leçons précédentes ont déjà introduit le concept de contenu ouvert, en se concentrant sur les licences Creative Commons. Outre les licences Creative Commons, il existe d'autres licences de contenu ouvert, dont certaines concernent un contenu spécifique et sont également régulièrement utilisées pour les "sous-produits" (tels que la documentation) des projets open source.

Licences pour la documentation (logicielle)

Les grands dépôts de logiciels contiennent régulièrement, outre le code, de la documentation ou des manuels relatifs au logiciel. La documentation contient généralement divers éléments qui peuvent être protégés par le droit d'auteur, tels que des textes explicatifs, des illustrations graphiques ou des exemples de code. Les licences open source qui s'appliquent au contenu d'un référentiel s'appliquent généralement aussi à ces textes si aucune autre information n'est disponible ; par exemple, la GPLv3.0 s'applique à "toute œuvre qui peut être protégée par le droit d'auteur" et ne se limite pas au code.

Cependant, la documentation est souvent soumise à des conditions de licence différentes. Cela peut se justifier pour diverses raisons, car la documentation est produite et utilisée différemment

du code. Les conditions d'une licence copyleft stricte pour les programmes d'ordinateur peuvent, par exemple, empêcher l'adoption de parties individuelles de la documentation pour d'autres programmes. La signification de l'expression "code source" pour la documentation sous licence GPLv3.0 n'est pas claire non plus. En outre, il était courant de distribuer des versions imprimées de la documentation ou des manuels des programmes. Cette situation peut être mieux prise en compte par des licences de documentation spécifiques (par exemple, si certaines obligations de licence sont liées à un certain nombre de copies distribuées).

Les licences Creative Commons sont souvent utilisées pour la documentation, mais il existe également des licences copyleft spéciales pour la documentation, comme la *Licence de documentation libre* (*Free Documentation License* - FDL), qui explique son objectif dans son préambule :

L'objectif de cette licence est de rendre un manuel, un guide ou tout autre document fonctionnel et utile « libre » au sens de la liberté : assurer à chacun la liberté effective de le copier et de le redistribuer, avec ou sans modification, que ce soit à des fins commerciales ou non commerciales. En second lieu, cette licence préserve pour l'auteur et l'éditeur un moyen d'obtenir le crédit de leur travail, tout en n'étant pas considéré comme responsable des modifications effectuées par d'autres.

Cette licence est une sorte de "copyleft", ce qui signifie que les travaux dérivés du document doivent eux-mêmes être libres dans le même sens. Elle complète la licence publique générale GNU, qui est une licence avec copyleft conçue pour les logiciels libres.

Nous avons conçu cette Licence afin de l'utiliser pour les manuels de logiciels libres, car les logiciels libres ont besoin d'une documentation libre : un programme libre devrait être accompagné de manuels offrant les mêmes libertés que le logiciel. Mais cette Licence n'est pas limitée aux manuels de logiciels ; elle peut être utilisée pour toute œuvre textuelle, quel que soit son sujet ou qu'elle soit publiée sous forme de livre imprimé. Nous recommandons cette licence principalement pour les ouvrages dont le but est l'instruction ou la référence.

— Free Documentation License, Préambule (traduction non officielle)

La licence est très spécifique à la documentation ; par exemple, elle concerne également les versions imprimées. En particulier, une licence copyleft de documentation n'affecte pas explicitement le code avec lequel elle est livrée. Bien qu'il soit également possible de distribuer une documentation sous GPLv3.0 avec un code sous licence MIT, dans ce cas, il s'agirait de deux œuvres indépendantes : Comme il s'agirait dans ce cas de deux œuvres indépendantes, la GPLv3.0 de la documentation n'entraînerait pas l'obligation pour le code d'être sous GPLv3.0. Mais une licence dédiée à la documentation apporte plus de clarté à cet égard.

Licences pour bases de données

Il existe également des licences spéciales pour les bases de données. Ces licences tiennent compte, entre autres, de l'utilisation d'une base de données comme ressource pour un programme et des particularités de la protection du droit d'auteur pour les bases de données.

À quoi s'applique le terme “base de données”

En matière de droit d'auteur, certains systèmes juridiques (y compris en Europe) font une distinction entre les bases de données qui, comme les autres œuvres protégées par le droit d'auteur (images, textes, etc.), bénéficient d'une protection sur la base d'une *création intellectuelle personnelle* (sélection et disposition d'éléments), et les bases de données dont la production (acquisition, vérification, présentation) a nécessité un *investissement substantiel* (droit sur les bases de données).

Dans les deux cas, l'objet de la protection est un ensemble d'éléments indépendants (tels que des œuvres distinctes ou des éléments de données) qui sont disposés de manière systématique ou méthodique et individuellement accessibles par voie électronique ou par d'autres moyens. La possibilité de protéger par le droit d'auteur les éléments eux-mêmes n'est pas importante, de sorte qu'il peut aussi s'agir de simples valeurs numériques.

Une collection de données peut donc être protégée par le droit d'auteur en tant qu'œuvre de base de données ou par le droit des bases de données (dans l'UE). Pour qu'une base de données soit protégée, la disposition et la sélection des éléments doivent être l'expression d'une création intellectuelle personnelle. Il est peu probable que ce soit le cas pour les collections de données qui servent de base à un logiciel — mais celles-ci peuvent être protégées par le droit des bases de données. Pour obtenir ce droit, un investissement substantiel doit avoir été réalisé dans la collecte et la disposition des éléments. Cet investissement peut également consister en temps et en ressources humaines.

Il existe aujourd'hui différentes licences pour les bases et les collections de données qui couvrent mieux l'utilisation des données que les licences “classiques” des logiciels libres. Examinons cette distinction à l'aide d'un exemple.

Dans le cadre d'un projet de recherche universitaire, tous les poèmes d'un auteur doivent être rassemblés et publiés sur un site web, où ils sont classés par ordre alphabétique et peuvent être consultés individuellement. Toutefois, le projet exige que les droits d'auteur pertinents pour la distribution et la reproduction des poèmes soient obtenus auprès de l'auteur. Des droits de licence sont payés à cet effet.

Les droits de licence peuvent constituer un “investissement substantiel”. Étant donné que les

poèmes peuvent être consultés individuellement et qu'ils sont classés systématiquement (selon l'alphabet), la base de données contenant les poèmes peut être protégée en vertu du droit des bases de données. Cependant, les éléments (poèmes) n'ont pas été spécifiquement sélectionnés (la sélection a été faite sur la base de l'exhaustivité) et l'arrangement n'est pas créatif, mais purement systématique. Il n'y a donc pas de création intellectuelle personnelle en relation avec la base de données, de sorte que la base de données ne peut être qualifiée d'œuvre de base de données.

Si, en revanche, les poèmes ont été arrangés et sélectionnés selon certains critères — par exemple pour documenter certains motifs de l'œuvre de l'auteur dans le contexte de la période de création — une création intellectuelle personnelle pourrait exister, et ce recueil de poèmes pourrait alors être protégé en tant qu'œuvre de base de données.

Toutefois, les deux catégories sont traitées différemment dans le cadre du droit d'auteur, notamment en ce qui concerne la durée de protection : les bases de données relevant du droit des bases de données ne sont protégées que 15 ans après leur production/publication, contre 70 ans après la mort de l'auteur pour les œuvres de base de données.

Délimitation pour les données

Les *données* contenues (dans notre exemple, les poèmes) doivent toujours être considérées séparément de la base de données dans son ensemble, car elles peuvent (mais ne doivent pas nécessairement) être éligibles à la protection en tant que telles et peuvent donc être soumises à des conditions de licence différentes. Par exemple, la collection de poèmes peut être soumise à la licence Open Database License (voir ci-dessous), tandis que les poèmes individuels peuvent être soumis à une licence Creative Commons ou à une licence propriétaire.

Délimitation pour les systèmes de gestion de bases de données (SGBD)

En outre, le logiciel de gestion des données, appelé *Système de Gestion de Base de Données* (SGBD), qui n'est pas considéré comme faisant partie de la base de données en vertu de la législation sur le droit d'auteur, mais comme un moyen d'accéder aux données, doit être considéré séparément en vertu de la législation sur le droit d'auteur. Ces applications — telles que MySQL, PostgreSQL, MariaDB et autres — doivent être considérées séparément comme des logiciels et peuvent être protégées en tant que telles.

Ensembles de données pour l'entraînement des modèles d'apprentissage automatique

Diverses collections de données peuvent être téléchargées à partir de sites web tels que huggingface.co ou kaggle.com, par exemple pour entraîner des modèles d'apprentissage automatique. Ces collections, également appelées *ensembles de données*, sont généralement soumises au droit des bases de données, à condition qu'un "investissement substantiel" ait été

réalisé pour leur production (ce qui n'est généralement pas évident de l'extérieur).

Le terme “ensemble de données” est quelque peu vague car il peut faire référence à une ligne dans une base de données — c'est-à-dire une entrée ou un élément définissable — mais aussi à des collections de données ou d'ensembles de données.

Open Database License (ODbL)

Ces distinctions sont également visibles dans la *Open Database License* (ODbL), une licence largement utilisée dans le secteur des bases de données. L'ODbL a été développée par Open Data Commons, un projet de l'Open Knowledge Foundation. Dans son préambule, la licence établit une distinction claire entre la protection de la base de données et la protection du contenu individuel de la base de données. La licence ne se réfère qu'à la première, tandis que le contenu peut également faire l'objet d'une autre licence indépendamment de celle-ci.

Bien entendu, il est toujours possible de choisir la même licence pour la base de données et le contenu, par exemple en compilant une collection d'œuvres sous licence CC-BY-4.0 dans une base de données elle-même sous licence CC-BY-4.0.

La meilleure licence pour la base de données dépend des objectifs du producteur de la base de données. Reprenons notre exemple de recueil de poèmes : Si la base de données de poésie doit être modifiée à volonté et peut également être distribuée sous d'autres conditions de licence, l'ODbL n'est pas le bon choix. En effet, l'ODbL contient un copyleft pour les bases de données, ce qui signifie qu'une base de données dérivée d'une base de données sous licence ODbL (par exemple, une base de données qui adopte tous les éléments et la disposition des éléments de la base de données originale et en ajoute d'autres) ne peut être redistribuée que sous les mêmes conditions. Dans ce cas, une licence Creative Commons avec droits d'édition (par exemple CC-BY) est un meilleur choix.

L'ODbL précise également qu'elle ne s'applique en aucun cas aux programmes informatiques qui gèrent la base de données (c'est-à-dire un SGBD) : “La présente licence ne s'applique pas aux programmes informatiques utilisés dans la création ou l'exploitation de la base de données”.

Toutefois, la licence stipule qu'une œuvre créée à partir du contenu de la base de données et disponible pour un usage public (une *œuvre produite*) doit au moins indiquer la base de données (et non pas le SGBD) qui a été utilisée et que la base de données elle-même peut être utilisée selon les conditions de l'ODbL. Un exemple d'une telle *œuvre produite* est une carte des rues créée à partir des coordonnées collectées dans la base de données.

Si l'une des licences copyleft connues (spécifiques aux logiciels), telle que la GPLv3.0, était utilisée pour une base de données, on peut supposer qu'un programme informatique utilisant la base de données (par exemple, pour la gestion de stock) ne pourrait être distribué que sous les termes de

la GPLv3.0. La licence d'un SGBD, en revanche, est susceptible d'être indépendante de la licence de la base de données, car le SGBD n'est pas considéré comme une "œuvre dérivée" de la base de données ; il permet simplement l'accès et l'édition et constitue donc une œuvre indépendante.

Community Data License Agreements (CDLA)

L'*Accord de Licence de Données Communautaires (Community Data License Agreement - CDLA)*, un projet de la Fondation Linux, se focalise également sur l'octroi de licences pour les données (par opposition aux logiciels). Les collections de données destinées à l'apprentissage automatique peuvent également faire l'objet d'un CDLA.

Nos communautés souhaitent développer des accords de licence de données qui permettraient un partage des données similaire à ce qui se fait avec les logiciels open source. Le résultat est une collaboration à grande échelle sur les licences de partage de données dans un cadre juridique que nous appelons : Community Data License Agreement (CDLA).

Ces licences établissent le cadre du partage collaboratif des données qui a fait ses preuves dans les communautés de logiciels open source.

— Site web CDLA (cdla.dev), Traduction non officielle

Le CDLA est donc basé sur un cadre qui prend en compte différents niveaux de licences, en particulier lorsque les données sont compilées à partir de sources multiples par différents contributeurs. Dans notre exemple, il pourrait s'agir d'une collection dans laquelle différents auteurs contribuent par leurs propres poèmes à la constitution d'une base de données commune sur la poésie.

Le CDLA fait la distinction entre la "licence entrante" ("inbound license") pour les données qui sont ajoutées à la collection et la "licence sortante" ("outbound license") pour l'utilisation des données. À un troisième niveau, il spécifie les exigences pour l'organisation qui héberge ou détient les données.

Pour en rester avec l'exemple de la poésie : Un auteur qui apporte son poème sera guidé par les spécifications des "licences entrantes". La "licence sortante" concerne les conditions convenues par les éditeurs du recueil de poésie dans son ensemble.

À l'instar des licences open source pour les logiciels, les CDLA sont disponibles dans les catégories *permissive* et *partage (sharing)*, que nous allons brièvement présenter. Une visite sur les sites web kaggle.com ou huggingface.co mentionnés ci-dessus vous donnera une idée des licences utilisées pour les bases de données ou les ensembles de données : Par exemple, vous pouvez utiliser des filtres pour savoir quelles collections de données sont proposées sous CDLA.

Community Data License Agreement — Permissive

La version 2.0 de la licence CDLA permissive est disponible depuis 2021. Elle est beaucoup plus concise et clairement formulée que la version 1.0.

La version 1.0 accorde le droit d'utiliser et de publier les données sous licence, y compris la publication de ce que l'on appelle les "données enrichies", qui comprennent également les modifications ou les ajouts apportés à l'ensemble de données. Lors de la transmission ou de la publication des données, les obligations de base de la licence doivent être respectées ; entre autres, le texte de la licence doit être transmis, les modifications doivent être signalées et les mentions du droit d'auteur doivent être conservées. Les données peuvent aussi être transmises sous des conditions de licence supplémentaires ou différentes.

Dans la version 2.0, l'expression "données enrichies" n'est plus utilisée ; à la place, l'octroi des droits est formulé de manière plus précise :

Un récepteur de données peut utiliser, modifier et partager les données mises à disposition par le(s) fournisseur(s) de données dans le cadre du présent accord, à condition que ce récepteur de données respecte les conditions du présent accord.

— Community Data License Agreement - Permissive, version 2.0 (traduction non officielle)

La seule condition pour partager les données dans la version 2.0 est la transmission du texte de la licence ou d'un lien vers le texte de la licence.

Community Data License Agreement — Sharing

La version partage du CDLA, qui n'est actuellement disponible qu'en version 1.0, contient un copyleft pour les données de telle sorte que le transfert ("publier") des "données enrichies" n'est autorisé que dans les mêmes conditions.

Les données (y compris les données enrichies) doivent être publiées en vertu du présent accord conformément à la section 3.

— Community Data License Agreement -- Sharing, version 1.0 (traduction non officielle)

Libre accès

Un terme souvent utilisé en rapport avec le contenu ouvert est "libre accès". Bien que ce terme ne soit pas défini au sens juridique, la déclaration de l'Initiative de Budapest pour le libre accès (*Budapest Open Access Initiative* - BOAI) clarifie l'intention qui sous-tend le libre accès. Cette initiative est née d'une conférence tenue à Budapest en 2001 et résume les efforts internationaux en faveur du libre accès :

La littérature qui devrait être accessible en ligne gratuitement est celle que les savants donnent au monde sans en attendre de rétribution. Principalement, cette catégorie englobe leurs articles dans des revues à comités de lecture, mais aussi toute prépublication n'ayant pas encore fait l'objet d'une évaluation qu'ils pourraient souhaiter mettre en ligne pour commentaire ou pour avertir leurs collègues d'une découverte scientifique importante. Il existe de nombreux degrés et types d'accès plus large et plus simple à cette littérature. Par "accès libre" à cette littérature, nous entendons sa mise à disposition gratuite sur l'Internet public, permettant à tout un chacun de lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces articles, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale, sans barrière financière, légale ou technique autre que celles indissociables de l'accès et l'utilisation d'Internet. La seule contrainte sur la reproduction et la distribution, et le seul rôle du copyright dans ce domaine devrait être de garantir aux auteurs un contrôle sur l'intégrité de leurs travaux et le droit à être correctement reconnus et cités.

— Déclaration de l'Initiative de Budapest pour le libre accès

Le libre accès désigne donc en particulier le libre accès à la littérature scientifique et à d'autres contenus sur Internet. Le mouvement pour le libre accès a vu le jour dans les années 1990 dans le but de rendre les publications scientifiques accessibles au grand public. Le libre accès ne fait donc pas référence à une licence spécifique ou à certaines conditions de licence, mais plutôt à un concept de licence.

Lorsqu'un article est publié en libre accès, la première étape consiste à choisir une licence de contenu libre, telle que l'une des licences Creative Commons. Cependant, le choix d'une licence CC n'est pas obligatoire pour le libre accès. Au fil des ans, une catégorisation des différentes stratégies de libre accès s'est développée, qui tient également compte de la double licence — par exemple en accordant une licence complète à un éditeur tout en proposant simultanément le contenu en téléchargement sur son propre site web.

La BOAI et la *Déclaration de Berlin*, qui a suivi deux ans plus tard, constituent toutes deux des étapes importantes dans le mouvement pour le libre accès. Le préambule de la déclaration de Berlin stipule :

Conformément à l'esprit de la Déclaration de Budapest pour l'accès ouvert, de la charte ECHO et de la Déclaration de Bethesda pour l'édition en libre accès, nous avons élaboré la déclaration de Berlin pour promouvoir un Internet qui soit un instrument fonctionnel au service d'une base de connaissance globale et de la pensée humaine, et de définir des mesures qui sont à envisager par les responsables politiques en charge de la science, les institutions de recherche, les agences de financement, les bibliothèques, les archives et les musées.

— Déclaration de Berlin 2003

Exercices guidés

1. Les licences open source “classiques” peuvent-elles également être utilisées pour la documentation et les bases de données ?

Oui	
Non	
Cela dépend, il n’y a pas de réponse précise	

2. Est-il judicieux d’utiliser des licences distinctes pour la documentation ? Pourquoi ?

3. Est-il judicieux d’utiliser des licences distinctes pour les bases de données ? Pourquoi ?

4. Parmi les licences suivantes, lesquelles conviennent pour la documentation ?

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

5. Que signifie le terme “libre accès” ?

Exercices d'exploration

1. Existe-t-il des licences spécifiques pour les polices de caractères ?

2. Expliquez brièvement deux stratégies ou modèles d'accès libre.

3. Que signifie “définition ouverte” ?

Résumé

Outre les licences Creative Commons bien connues, il existe d'autres concepts de licence de contenu encore plus spécifiquement adaptés aux types de données concernés : pour la documentation logicielle ou les bases de données, il existe les licences FDL ou ODbL qui tiennent compte des caractéristiques de ces types d'œuvres. Cette leçon donne un aperçu de certaines de ces licences et aborde également le concept de libre accès.

Réponses aux exercices guidés

1. Les licences open source “classiques” peuvent-elles également être utilisées pour la documentation et les bases de données ?

Oui	X
Non	
Cela dépend, il n’y a pas de réponse précise	

2. Est-il judicieux d’utiliser des licences distinctes pour la documentation ? Pourquoi ?

Oui, car elles contiennent généralement d’autres informations que le programme lui-même. Elles régissent également la manière dont les impressions de la documentation doivent être traitées, par exemple.

3. Est-il judicieux d’utiliser des licences distinctes pour les bases de données ? Pourquoi ?

Oui, cela peut être utile. En particulier, une licence spécifique pour les bases de données peut permettre d’obtenir un effet copyleft pour la base de données sans que le copyleft n’affecte le code environnant. En outre, les licences pour les bases de données traitent des dispositions spécifiques prévues par la loi sur le droit d’auteur pour la protection des bases de données..

4. Parmi les licences suivantes, lesquelles conviennent pour la documentation ?

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

5. Que signifie le terme “libre accès” ?

Ce terme désigne le libre accès à la littérature scientifique et à d’autres contenus sur Internet.

Réponses aux exercices d'exploration

1. Existe-t-il des licences spécifiques pour les polices de caractères ?

Oui, il existe par exemple la licence SIL Open Font License (OFL), qui contient un copyleft spécifique aux polices de caractères, mais celle-ci n'a aucun effet si les polices sont utilisées sans modification.

2. Expliquez brièvement deux stratégies ou modèles d'accès libre.

Dans la stratégie d'accès libre "Or" ("*Gold*" *open access strategy*), la publication est mise à disposition directement par l'éditeur sous une licence de contenu libre, généralement (mais pas exclusivement) sous une licence Creative Commons.

Le modèle d'accès libre "Vert" ("*Green*" *open access model*) s'applique lorsque l'éditeur ne distribue pas la publication en accès libre, mais que l'auteur a la possibilité de la mettre à disposition en téléchargement, par exemple sur son site web sous une licence libre.

3. Que signifie "définition ouverte" ?

La "définition ouverte" est une interprétation commune, formulée par l'Open Knowledge Foundation, du terme "ouvert" dans les expressions "données ouvertes" ("*_open data_*"), "connaissances ouvertes" ("*_open knowledge_*") et "contenu ouvert" ("*_open content_*"). La définition décrit en particulier les libertés fondamentales qui doivent être accordées pour qu'une licence soit "ouverte" au sens de la définition. Il s'agit notamment de l'accès, de l'utilisation, de l'édition ou de la modification et du partage.



Thème 054: Modèles d'entreprise des logiciels open source



054.1 Modèles d'entreprises de développement logiciel

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 054.1

Valeur

2

Domaines de connaissance les plus importants

- Comprendre les objectifs et les raisons de publier un logiciel ou un contenu sous une licence ouverte
- Comprendre les modèles d'entreprise usuels et les sources de revenus des organisations qui développent des logiciels open source et du contenu ouvert
- Comprendre les implications de l'utilisation de logiciels open source en tant que composants de produits et services technologiques plus importants
- Comprendre l'impact des licences sur les modèles d'entreprise de développement de logiciels
- Comprendre les considérations relatives aux logiciels open source du point de vue du client
- Sensibilisation aux structures de coûts et aux investissements nécessaires pour les modèles d'entreprise de développement de logiciels open source

Liste partielle de termes, fichiers et utilitaires utilisés

- Développement payant
- Cœur ouvert (open-core) et modules complémentaires payants
- Gratuit-payant (Freemium)
- Versions entreprise et communautaire
- Distribution auto-hébergée

- Abonnements
- Support client



Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	054 054 Modèles d'entreprises open source
Objectif :	054.1 Modèles d'entreprises de développement logiciel
Leçon :	1 sur 1

Introduction

Traditionnellement, les entreprises qui distribuaient des logiciels propriétaires (également appelés logiciels à *code source fermé*) avaient un modèle commercial relativement simple : elles vendaient une licence d'utilisation du logiciel, soit sous forme de paiement unique, soit sous forme d'un *abonnement*.

Mais au fil des décennies, le marché des logiciels a radicalement changé, et les clients s'attendent aujourd'hui à payer quelques dollars pour une application et à bénéficier de mises à jour gratuites à vie.

En conséquence, l'ancienne approche des logiciels propriétaires n'est plus rentable, et même les entreprises qui avaient l'habitude de baser l'ensemble de leur modèle commercial sur la vente de licences logicielles se sont diversifiées dans l'intégration de fournisseurs, l'assistance, les services, les logiciels en tant que service (*Software-as-a-Service* - SaaS), l'hébergement cloud/conteneur et les produits matériels qui exécutent des logiciels (téléphones, ordinateurs portables, serveurs, imprimantes, voitures, montres intelligentes, etc.).

En revanche, les licences de logiciels libres et open source donnent à quiconque le droit d'utiliser, d'étudier, de modifier et de redistribuer le logiciel sans payer de frais. Ainsi, les entreprises basées sur l'open source n'ont jamais eu la possibilité de vendre une licence d'utilisation du logiciel. Vous ne devez pas vous attendre à pouvoir simplement reprendre le travail d'autres personnes issu d'un projet open source et réaliser d'énormes profits en proposant ce travail accessible au public à vos clients, car ceux-ci peuvent obtenir le même logiciel gratuitement directement depuis le site du projet. Au lieu de cela, les entreprises qui fournissent des logiciels libres et open source utilisent des modèles commerciaux alternatifs. Les modèles commerciaux les plus performants dans le domaine des logiciels combinent souvent plusieurs formes de valeur ajoutée pour le client.

Cette leçon examine les raisons pour lesquelles une entreprise choisirait certains de ces modèles commerciaux avec des logiciels open source, ainsi que les compromis inhérents à ces options.

Objectifs et raisons de publier un logiciel ou un contenu sous une licence libre

Il existe de nombreuses raisons pour lesquelles les développeurs choisissent de créer une entreprise basée sur le code open source : le logiciel peut répondre à un besoin, résoudre un problème ou fournir certaines fonctionnalités que les développeurs proposent à leurs clients. Une entreprise peut gagner du temps et de l'argent en partageant le travail sur le logiciel avec d'autres développeurs, plutôt que de réécrire le même logiciel à partir de zéro. Participer au projet est un investissement qui, on l'espère, sera rentable sous la forme d'un logiciel utilisable, stable, fiable, sécurisé et bien entretenu.

De nombreux développeurs open source espèrent obtenir des corrections de bogues de la part de leurs clients. Certains clients vont encore plus loin dans leurs contributions en ajoutant de nouvelles fonctionnalités, en portant le logiciel vers un nouvel environnement ou en apportant d'autres améliorations significatives. Les corrections de bogues et les améliorations de haut niveau améliorent le logiciel d'origine. Si les clients les reversent aux développeurs d'origine, ces modifications peuvent rendre le code plus attrayant pour d'autres clients potentiels et encourager son adoption.

Mais une entreprise ne peut pas s'attendre à recevoir des contributions et de l'attention simplement en plaçant son code sur un site tel que GitHub ou GitLab. Encourager les développeurs à rejoindre un projet est une chose, mais les maintenir enthousiastes en est une autre — il ne faut donc pas sous-estimer la quantité de travail que requièrent les contributions de la communauté. Une entreprise doit s'efforcer de recruter des développeurs, de les encadrer, de les motiver et de vérifier leur code afin d'en détecter les éventuelles failles.

Une autre raison d'ouvrir le code est de créer une norme industrielle. Le partage ouvert du code

pourrait améliorer l'interopérabilité et apporter d'autres avantages. L'équipe qui a développé le code possède une expertise précieuse qui pourrait lui permettre d'orienter l'avenir du projet et d'être rémunérée pour aider les autres utilisateurs du code.

Certaines entreprises ouvrent leur code afin d'instaurer un climat de confiance auprès de leurs clients. Tout d'abord, les clients savent qu'ils peuvent continuer à utiliser et à améliorer le code si l'entreprise fait faillite ou décide de se lancer sur un autre marché et abandonne le code. Ensuite, les clients peuvent vérifier eux-mêmes la qualité et la sécurité du code ou demander à un expert indépendant de réaliser un audit, ce qui n'est généralement pas possible avec les logiciels propriétaires fermés.

Enfin, une entreprise peut avoir adopté un code source déjà open source dans le but de développer une activité commerciale à partir de celui-ci. La licence peut alors exiger que l'entreprise distribue le code source de ses modifications à tous les utilisateurs finaux du programme.

En résumé, voici quelques raisons d'ouvrir le code d'une entreprise :

- S'appuyer sur un projet de logiciel libre existant
- Tirer parti de l'innovation et des contributions des clients
- Instaurer la confiance auprès des clients
- Promouvoir le code en tant que norme industrielle

Modèles d'entreprises courants et sources de revenus

Au cours des dernières décennies, l'industrie du logiciel a découvert de nombreux modèles commerciaux adaptés aux entreprises propriétaires et open source. Cette section se concentre sur les modèles les plus populaires actuellement utilisés par les développeurs open source.

Un modèle commercial courant, tant pour les logiciels propriétaires que pour les logiciels open source, consiste à facturer l'assistance. Les clients peuvent rencontrer des difficultés pour installer et configurer le logiciel, corriger les bogues au fur et à mesure qu'ils apparaissent, ajouter de nouvelles fonctionnalités à leur usage exclusif et gérer leurs systèmes. L'équipe qui a développé le logiciel est une excellente source d'assistance. En fait, cette équipe offre probablement déjà une assistance gratuite sur les forums où le logiciel est discuté.

Ainsi, dans le modèle d'assistance, un client paie un fournisseur pour installer le logiciel open source sur son matériel (ce qu'on appelle une *distribution auto-hébergée* ou, en anglais, *self-hosting distribution*) ou obtient l'accès au logiciel sur le matériel du fournisseur (modèle basé sur le cloud). Les contrats d'assistance garantissent aux clients une aide rapide de la part de

l'entreprise pour leurs besoins plus complexes. Les entreprises qui utilisent ce modèle proposent généralement des contrats de services, que les clients paient régulièrement.

Un autre modèle commercial pour les logiciels propriétaires — en réalité un ensemble de modèles qui se recoupent — est appelé *gratuit-payant (freemium)*, un terme popularisé en 2009 par l'auteur et éditeur Chris Anderson, et basé sur un modèle commercial beaucoup plus ancien connu sous le nom de *rasoir et lames*. Dans ce modèle, les clients paient très peu pour le produit de base (par exemple, un rasoir ou une imprimante), puis paient un supplément pour les composants nécessaires conçus pour s'user (lames pour le rasoir, cartouches d'encre pour l'imprimante).

Dans le modèle freemium, les clients peuvent utiliser un produit à un certain niveau gratuitement et sont encouragés à payer pour bénéficier de fonctionnalités supplémentaires s'ils trouvent le produit utile. Vous connaissez probablement des sites d'actualités en ligne qui proposent un certain nombre d'articles gratuits et demandent aux lecteurs de s'abonner pour accéder à l'intégralité de leur contenu ; il s'agit là d'un exemple bien connu de modèle freemium. Un autre exemple est celui d'une plateforme de jeux qui propose le jeu de base gratuitement, mais facture certains éléments spécifiques.

D'autres éditeurs de logiciels propriétaires basent leur modèle freemium sur le temps : essayez le logiciel gratuitement pendant trois mois, puis payez pour continuer à l'utiliser.

Les éditeurs de logiciels propriétaires utilisent parfois une variante du modèle freemium appelée *cœur ouvert (open core)*. Dans ce modèle, certaines fonctionnalités de base (le "cœur" du produit) sont open source, tandis que des fonctionnalités propriétaires supplémentaires peuvent être concédées sous licence.

Souvent, la partie ouverte est pleinement fonctionnelle, mais elle convient mieux aux chercheurs ou aux utilisateurs individuels, et devient difficile à gérer lorsque de nombreuses personnes la partagent au sein d'une entreprise. Par conséquent, les fonctionnalités propriétaires peuvent inclure des interfaces web pratiques pour l'administration, des outils de comptabilité et de collaboration, ainsi que d'autres éléments présentant un intérêt particulier pour les grands sites.

Même si les modèles commerciaux open core utilisent certains logiciels open source, les clients sont suffisamment avisés pour reconnaître que le produit dans son ensemble est en réalité propriétaire. Ainsi, si l'open core présente l'avantage de s'appuyer sur un projet open source existant, il ne procure aucun autre avantage propre aux logiciels open source. Un modèle commercial open core ne permet pas d'établir une relation de confiance avec les clients, de les inciter à contribuer au logiciel de base, de leur donner accès au code, de créer un réseau de développeurs compétents en dehors de l'entreprise principale ou de servir de norme industrielle. Pire encore, le modèle de développement open core entraîne les mêmes coûts de développement, sans offrir les avantages qui le rendent intéressant. Les entreprises qui tentent un modèle

commercial open core sont généralement déçues des résultats et beaucoup d'entre elles passent ensuite à un modèle purement propriétaire.

Les entreprises peuvent également créer un service web à partir d'un code source ouvert ou propriétaire, selon le modèle *Software as a Service* (SaaS). Les clients paient alors un abonnement mensuel ou annuel.

De nombreuses entreprises qui fournissent des services web SaaS ou proposent des applications mobiles tirent leurs revenus de la publicité. Certaines entreprises collectent également des données sur les utilisateurs via leur service ou leur application et vendent ces données à des tiers qui peuvent les utiliser à des fins publicitaires. Les publicités peuvent toutefois être considérées comme gênantes. La vente de données est controversée, et certains pays imposent des restrictions sur la collecte de données.

Enfin, les entreprises peuvent développer et commercialiser des logiciels libres sans chercher à en tirer le moindre profit. Ce modèle est accessible aux entreprises qui tirent leurs revenus d'autres sources que les logiciels. Par exemple, les constructeurs automobiles collaborent pour créer de nombreux logiciels libres destinés à fonctionner dans leurs voitures (qui sont aujourd'hui entièrement informatisées).

Les grandes entreprises de logiciels qui tirent leurs revenus de produits propriétaires, telles que Google et Amazon, publient parfois des logiciels administratifs ou d'autres outils utiles en open source, car ces outils open source ne sont pas essentiels à leur activité principale. Les organisations qui publient le code sous une licence open source bénéficient des commentaires, des rapports de bogues et des améliorations de fonctionnalités fournis par la communauté open source.

Utilisation de logiciels open source dans d'autres technologies et services

De nombreuses entreprises intègrent des logiciels open source dans leurs produits ou leurs plateformes web. Après tout, les logiciels open source sont gratuits ! Mais ce n'est pas la raison la plus importante (et peut-être même pas une bonne raison) pour les adopter. Plus important encore, bon nombre de ces logiciels sont de très bonne qualité (même si l'équipe de développement doit les examiner attentivement avant de les adopter) et peuvent même constituer une norme industrielle.

Un autre avantage des logiciels open source est qu'ils resteront disponibles même si les développeurs ou la communauté qui les entoure disparaissent.

Mais les logiciels libres et open source impliquent certaines responsabilités. Cette section présente

rapidement les principaux points à prendre en compte avant de les adopter.

La première question s'applique à tout logiciel ou outil tiers dont l'adoption est envisagée : répondra-t-il aux besoins des utilisateurs aujourd'hui et à mesure que l'entreprise évoluera ? Le logiciel est-il soutenu par une communauté solide capable d'offrir une assistance technique et de continuer à le développer ? Le logiciel présente-t-il des failles de sécurité importantes ?

Ensuite, les dirigeants doivent réfléchir aux responsabilités de l'entreprise si elle intègre un projet open source dans son propre logiciel. Si les développeurs créent un nouveau code à partir du code open source, ils doivent vérifier ce qu'ils doivent faire selon la licence du logiciel open source. Certaines licences demandent aux développeurs de distribuer le code source de leurs propres modifications à leurs utilisateurs finaux sous la même licence libre que le code source original.

Même si un développeur n'est pas tenu de distribuer son code source modifié et crée un produit propriétaire à partir du code open source, il peut souhaiter apporter certaines contributions, telles que des corrections de bogues et des améliorations au code open source. En apportant ces corrections et améliorations, le développeur contributeur permet au projet de les intégrer (si les développeurs principaux le souhaitent) et de les maintenir. Les développeurs qui ne contribuent pas à apporter des améliorations pourraient devoir réappliquer les corrections et les améliorations à chaque fois qu'ils passent à une nouvelle version du code original.

En raison de cette dépendance, et pour d'autres raisons, le personnel de l'entreprise devrait envisager de devenir des membres actifs de la communauté qui développe le code. Les développeurs peuvent apprendre beaucoup sur le code en participant, et peuvent aider à définir l'orientation des développements futurs. Bien sûr, l'organisation devrait rémunérer le temps que les développeurs consacrent aux activités communautaires. Pour une entreprise qui souhaite sérieusement utiliser des logiciels libres, cela a un coût.

Considérations relatives aux logiciels open source du point de vue du client

L'ouverture du code source présente plusieurs avantages pour les clients, mais implique une relation différente entre l'entreprise et ses clients. Les clients doivent comprendre les avantages ainsi que les implications de cette relation.

Le principal avantage pour le client, mentionné plus haut dans cette leçon, est qu'il peut avoir une plus grande confiance dans le logiciel. Il sait qu'il ne disparaîtra pas. De nombreuses entreprises propriétaires ferment leurs portes ou prennent soudainement une nouvelle direction, abandonnant leurs clients qui n'ont parfois que peu de temps pour migrer vers un produit qui ne leur plaît peut-être pas. Les logiciels open source, en revanche, ne dépendent d'aucune organisation en particulier. Si le projet est important, d'autres membres de la communauté le

poursuivront lorsque les développeurs d'origine se seront retirés.

Les clients peuvent également vérifier la qualité et la sécurité des logiciels open source. Ils peuvent déterminer la facilité avec laquelle ils peuvent ajouter leurs propres fonctionnalités ou les porter vers un nouvel environnement.

Comme le code source d'un projet open source est accessible à tous, un large éventail de développeurs peut se familiariser avec celui-ci. Dans le cadre d'un projet bénéficiant d'une communauté active, de nombreuses personnes apportent leur aide sur le forum. Il est souvent facile de recruter des personnes pour assurer le support du logiciel ou pour l'administrer et l'utiliser au sein de l'entreprise, car les nouveaux employés connaissent déjà le code.

Il est très rassurant pour les clients, qui dépendent du code pour leur fonctionnement quotidien, de savoir qu'ils peuvent corriger eux-mêmes un bug ou engager quelqu'un pour le faire. Un bug obscur qui n'affecte que quelques clients peut prendre des années à être corrigé par l'équipe de développement principale, ce qui est une source de frustration constante pour les utilisateurs de logiciels propriétaires. De plus, lorsque le code source est disponible, un bug et sa correction suggérée peuvent être identifiés plus rapidement.

Qu'en est-il de la relation entre l'entreprise et le client ? L'entreprise peut choisir d'établir une relation similaire à celle proposée par une entreprise classique de logiciels propriétaires, en fournissant au client des mises à jour régulières et un contrat d'assistance. Le client n'a jamais besoin de consulter le code source ni de rejoindre les forums communautaires.

Mais la plupart des clients pourraient tirer profit des opportunités uniques offertes par les projets open source. Ceux-ci permettent à leurs propres développeurs de contribuer à la fois en fournissant des informations et en partageant du code. Une telle participation approfondit les connaissances de leur équipe, les aide à recruter de nouveaux collaborateurs et leur donne une influence sur les développements futurs.

Structures de coûts et investissements

Les programmeurs sont très recherchés, donc tout projet logiciel va coûter cher. Les personnes qui connaissent les principaux projets open source sont encore plus recherchées, car ces bases de code sont très largement utilisées.

Un projet open source offre des économies potentielles, car différentes organisations et personnes peuvent unir leurs efforts autour d'une base de code commune. Mais la gestion d'un tel projet engendre de nouveaux coûts.

Si une entreprise lance un projet développé en interne, elle doit investir pour qu'il puisse servir une communauté plus large (voire mondiale). Certaines fonctionnalités qui répondaient aux

besoins commerciaux spécifiques de l'entreprise devront peut-être être repensées afin de pouvoir servir d'autres entreprises également.

Si le code est médiocre ou mal conçu, une entreprise ne devrait probablement pas l'ouvrir. Les contributeurs potentiels ainsi que les clients potentiels seront découragés par la qualité. Il incombe aux développeurs de corriger ces problèmes de toute façon, car un logiciel mal codé est fragile : il est difficile à mettre à jour avec de nouvelles fonctionnalités et a tendance à développer des bogues complexes difficiles à corriger. Ces problèmes sont appelés *dette technique*, et plus tôt, ils sont corrigés, mieux c'est pour tous les utilisateurs.

Enfin, il est surprenant de constater à quel point les entreprises intègrent souvent des secrets commerciaux, des mots de passe, des références personnelles à des individus ou d'autres informations sensibles dans leur code. Les développeurs doivent consacrer du temps à éliminer ces vulnérabilités. Les mots de passe, les clés API, les certificats et les identifiants d'accès au cloud ne devraient de toute façon jamais figurer dans le code ; ils devraient être gérés en externe via un service sécurisé.

Supposons qu'une entreprise ait ouvert son code et espère bénéficier de contributions externes. Certains clients paieront les développeurs pour la maintenance et l'ajout de nouvelles fonctionnalités. D'autres affecteront leurs propres développeurs au projet, mais l'équipe de développement principale devra consacrer du temps à les aider. L'équipe de développement principale devra former les contributeurs externes au code et aux normes de développement associées. Elle devra également guider les contributeurs externes sur les éléments à ajouter et les endroits où envoyer leurs commentaires sur le code.

Gardez un œil sur les personnes extérieures qui font preuve de talent. Ces personnes pourraient être des recrues précieuses pour l'équipe principale. Elles pourraient souhaiter rejoindre l'équipe et apportent avec elles des connaissances considérables.

Lorsqu'une équipe de développeurs est rémunérée, tandis que d'autres personnes donnent bénévolement de leur temps, les bénévoles peuvent se sentir exploités ou se demander pourquoi ils devraient aider. Chaque contributeur, qu'il s'agisse d'un bénévole individuel ou d'une organisation, doit suivre le processus de réflexion décrit plus haut dans cette leçon afin de déterminer pourquoi il contribue.

Le code libre n'est pas gratuit, même s'il n'entraîne aucun frais de licence. Il s'agit simplement d'un modèle de développement différent.

Exercices guidés

1. Comment le code source ouvert améliore-t-il la confiance des clients dans le logiciel ?

2. Pourquoi les entreprises sont-elles tentées d'essayer un modèle commercial open core, et pourquoi ce modèle ne parvient-il pas à offrir les avantages des logiciels open source ?

3. Quels sont les types d'aide que les développeurs apportent généralement aux contributeurs externes à leurs projets open source ?

Exercices d'exploration

1. Vous envisagez de baser un produit propriétaire sur un projet open source. Quels facteurs prendriez-vous en considération pour prendre cette décision ?

2. Vous avez développé plusieurs produits, pour lesquels vous facturez un abonnement, à partir d'un projet open source. Que ferez-vous si la licence open source vous oblige à reverser tout votre code au projet ? De plus, vous avez ajouté la prise en charge de certains nouveaux protocoles de communication au projet open source afin de répondre à vos besoins spécifiques. Si vous avez le choix, demanderez-vous au projet open source d'intégrer cette prise en charge des protocoles dans son code de base ?

Résumé

Dans cette leçon, vous avez découvert l'intérêt de l'ouverture du code source. Vous avez passé en revue les modèles commerciaux couramment utilisés aujourd'hui et compris l'importance de bien connaître l'impact de la licence du code. Vous avez pris connaissance des questions à prendre en compte lors de l'intégration du code source ouvert dans votre entreprise et des avantages que l'open source apporte à vos clients. Vous avez également appris en quoi les coûts du développement open source diffèrent de ceux des logiciels propriétaires.

Réponses aux exercices guidés

1. Comment le code source ouvert améliore-t-il la confiance des clients dans le logiciel ?

Les clients peuvent vérifier la qualité et la sécurité du code. Ils peuvent également être assurés de pouvoir continuer à utiliser le code si les développeurs d'origine cessent de le prendre en charge.

2. Pourquoi les entreprises sont-elles tentées d'essayer un modèle commercial open core, et pourquoi ce modèle ne parvient-il pas à offrir les avantages des logiciels open source ?

À première vue, il semble que l'open core devrait offrir tous les avantages des logiciels open source, tout en permettant à une entreprise de continuer à utiliser un modèle commercial propriétaire qui vend une licence d'utilisation du logiciel. Dans la pratique, un modèle commercial open core ne parvient pas à offrir les avantages de l'open source, tout en conservant tous les coûts supplémentaires liés au développement ouvert.

3. Quels sont les types d'aide que les développeurs apportent généralement aux contributeurs externes à leurs projets open source ?

Les développeurs travaillant sur un projet forment et encadrent généralement les contributeurs externes, et vérifient que leurs contributions respectent les normes de qualité et de développement de l'équipe.

Réponses aux exercices d'exploration

1. Vous envisagez de baser un produit propriétaire sur un projet open source. Quels facteurs prendriez-vous en considération pour prendre cette décision ?

Tout d'abord, déterminez si le logiciel open source répond à vos besoins. Vérifiez sa qualité, les failles de sécurité qui ont été signalées publiquement et la santé de la communauté qui l'entoure. Lisez attentivement la licence pour voir si elle vous oblige à partager vos modifications du code. Déterminez dans quelle mesure vous souhaitez participer à la communauté du projet open source et comment vous allez définir les rôles que vos développeurs joueront au sein de cette communauté.

2. Vous avez développé plusieurs produits, pour lesquels vous facturez un abonnement, à partir d'un projet open source. Que ferez-vous si la licence open source vous oblige à reverser tout votre code au projet ? De plus, vous avez ajouté la prise en charge de certains nouveaux protocoles de communication au projet open source afin de répondre à vos besoins spécifiques. Si vous avez le choix, demanderez-vous au projet open source d'intégrer cette prise en charge des protocoles dans son code de base ?

Si le projet ne vous oblige pas à partager les modifications apportées au code, vous ne le ferez probablement pas, car il est plus facile de facturer un abonnement pour un produit propriétaire. Si vous devez partager votre code, proposez un abonnement à une distribution auto-hébergée. Même si vous n'avez pas besoin de partager votre code, vous souhaitez probablement demander au projet d'intégrer votre prise en charge des protocoles de communication afin de ne pas avoir à réimplémenter cette prise en charge à chaque fois que vous installez une nouvelle version du code open source.



054.2 Modèles d'entreprises de fournisseurs de services

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 054.2

Valeur

2

Domaines de connaissance les plus importants

- Comprendre les modèles d'entreprise et les sources de revenus habituels pour les organisations fournissant des services liés aux logiciels open source et au contenu ouvert
- Comprendre l'impact des licences sur les modèles d'affaires des fournisseurs de services
- Comprendre les objectifs et les accords de niveau de service
- Comprendre les besoins en matière de sécurité et de protection de la vie privée
- Sensibilisation aux structures de coûts et aux investissements nécessaires pour les modèles d'entreprise de services liés aux logiciels open source

Liste partielle de termes, fichiers et utilitaires utilisés

- Services hébergés
- Informatique en nuage
- Services de conseil
- Formation
- Vente de matériel
- Assistance aux utilisateurs
- Conditions générales d'utilisation (ToS)
- Objectifs de niveau de service (SLO)

- Accords de niveau de service (SLA)
- Accords sur le traitement des données



Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	054 Modèles d'entreprises open source
Objectif :	054.2 Modèles d'entreprises de fournisseurs de services
Leçon :	1 sur 1

Introduction

Les fournisseurs de services constituent la colonne vertébrale de toute entreprise ou organisation qui opère de quelque manière que ce soit via un réseau. Des fournisseurs d'accès à Internet (FAI), qui fournissent un accès à l'Internet mondial, aux applications spécialisées telles que la messagerie électronique ou la gestion de la clientèle, ce sont les services logiciels qui rendent possibles les opérations commerciales réelles.

Les logiciels open source font partie intégrante de nombreux modèles d'entreprises de fournisseurs de services. Par exemple, un système d'exploitation open source peut servir de base à une activité d'hébergement. Un outil d'orchestration open source pourrait être la colonne vertébrale d'une entreprise d'infrastructure en tant que service (*Infrastructure-as-a-Service* - IaaS) ou de plateforme en tant que service (*Platform-as-a-Service* - PaaS), telle qu'un fournisseur de cloud public ou une plateforme de conteneurs. Une application open source pourrait être fournie en ligne sous forme de logiciel en tant que service (*Software-as-a-Service* - SaaS). Certains fournisseurs de services proposent également des services complémentaires liés aux logiciels open source, tels que le conseil, la formation ou l'assistance à la clientèle.

Sources de revenus

Quand le service repose entièrement ou en grande partie sur des logiciels open source, on peut parler de *modèle d'entreprise open source*. Comme les logiciels open source peuvent être téléchargés et utilisés gratuitement, les modèles d'entreprises de services open source ont développé des produits et des services spécifiques en fonction, entre autres, de la proportion de logiciels libres dans le produit réel.

Si l'ensemble du logiciel est sous licence libre, un modèle d'entreprise peut consister en l'*hébergement*, c'est-à-dire la mise à disposition du logiciel sur une plateforme fiable et sécurisée. Cela évite aux clients d'avoir à gérer leurs propres serveurs, ce qui représente un effort et un risque. En contrepartie, les clients paient des frais d'utilisation au fournisseur de services. Ces frais peuvent être liés au nombre de comptes utilisateurs, au volume de données stockées ou transférées, ou à certains intervalles de temps. Dans certains cas, plusieurs fournisseurs différents proposent des services payants pour le même logiciel open source.

Ce modèle d'entreprise populaire implique souvent des *abonnements*, dans le cadre desquels les clients paient des frais mensuels ou annuels pour accéder au service. L'accès est généralement associé à des services ou des fonctionnalités supplémentaires. Les fournisseurs peuvent proposer plusieurs niveaux d'abonnement, facturant un tarif plus élevé pour davantage de fonctionnalités, un plus grand nombre d'utilisateurs, des services plus complets ou une garantie de fiabilité plus forte. Certains fournisseurs peuvent également proposer gratuitement un niveau de service de base, une variante du modèle "freemium".

Une autre source de revenus populaire pour le modèle d'entreprise de fournisseurs de services open source a toujours été d'offrir une *assistance*. L'assistance comprend des conseils, de la documentation et des formations pour les utilisateurs qui rencontrent des problèmes lors de l'utilisation du logiciel. Si le client préfère héberger le logiciel libre sur ses propres serveurs, l'installation en particulier peut souvent s'avérer difficile, car l'utilisateur peut avoir besoin d'installer d'autres outils et bibliothèques indispensables, ne pas savoir où les placer pour que tous les éléments fonctionnent ensemble, ou rencontrer des problèmes sur son matériel et son système d'exploitation particuliers. C'est pourquoi l'assistance et la formation fournies par un fournisseur de services sont précieuses.

Des *services supplémentaires*, tels que des fonctionnalités complémentaires ou liées à la sécurité, peuvent également être développés et fournis sur mesure pour chaque client. Ces fonctionnalités ou adaptations du logiciel aux besoins très spécifiques d'un client constituent un complément sophistiqué et lucratif au modèle d'entreprise. Il appartient alors au fournisseur de services et au client de décider si ces ajouts sont réintégrés dans le projet de base en tant que logiciel libre ou s'ils deviennent propriétaires.

Quelques projets open source proposent un *modèle de double licence*. Le logiciel est disponible sous une licence libre, qui répond aux besoins de nombreux utilisateurs. Cependant, certains utilisateurs souhaitent intégrer le logiciel dans un produit propriétaire, ce qui n'est pas possible sous une licence réciproque. C'est pourquoi une licence standard basée sur le droit d'auteur est proposée à ces utilisateurs. Le modèle de double licence est particulièrement efficace pour les bases de données, car de nombreuses entreprises souhaitent proposer une base de données riche en fonctionnalités et efficace dans le cadre d'un produit logiciel propriétaire.

Dans un *modèle de revenus publicitaires*, les clients ne paient pas pour accéder au service. Au lieu de cela, le fournisseur de services affiche des publicités aux clients qui utilisent le service et facture les entreprises pour placer leurs publicités. Les fournisseurs de services open source évitent souvent d'utiliser des services publicitaires propriétaires en raison de préoccupations liées à la vie privée des utilisateurs. Cependant, les services publicitaires open source alternatifs ne procèdent pas à un suivi publicitaire intrusif.

Dans un *modèle basé sur une commission*, le prestataire de services perçoit un pourcentage ou des frais pour la gestion des transactions via son service en ligne. Ce modèle est à la base de la plupart des sites de commerce électronique, des sites de réservation de voyages et des processeurs de paiement. De nombreux projets open source dans ce domaine sont des plateformes polyvalentes destinées à être soit hébergées par le prestataire de services, soit proposées comme service secondaire basé sur une commission, dans lequel le site web destiné aux clients paie un pourcentage ou des frais pour une version hébergée de la plateforme open source.

Un *modèle d'entreprise basé sur le contenu ouvert* consiste à créer et à distribuer du contenu sous des licences telles que Creative Commons afin que d'autres puissent librement l'utiliser, le modifier et le partager. Les utilisateurs sont encouragés à contribuer au contenu, afin de l'améliorer au fil du temps. Bien que le contenu lui-même soit gratuit, des revenus peuvent être générés grâce à des produits ou services complémentaires tels que des dons, l'affichage de publicités, l'offre de fonctionnalités freemium, la vente de produits dérivés ou la fourniture de services de conseil, de formation ou d'assistance.

Choisir entre les services hébergés et les services auto-hébergés

Les services hébergés sont idéaux pour les organisations qui recherchent une facilité d'utilisation, des coûts initiaux réduits, une évolutivité et une gestion professionnelle de la sécurité sans avoir besoin d'une expertise technique approfondie. Cependant, ces services présentent des inconvénients potentiels en termes de contrôle, de personnalisation, de confidentialité des données et de dépendance vis-à-vis du fournisseur.

Les services hébergés nécessitent généralement une configuration minimale, et le fournisseur de services se charge de la maintenance, des mises à jour et de la sécurité du système. Ils mettent

souvent à disposition des centres de données mondiaux, offrant ainsi de meilleures performances et une plus grande fiabilité aux utilisateurs internationaux. Les fournisseurs de services peuvent augmenter ou réduire les ressources en fonction de la demande, sans que le client ait à fournir beaucoup d'efforts. De plus, les services hébergés disposent souvent d'équipes de sécurité dédiées pour gérer et répondre aux menaces, et les fournisseurs de services peuvent offrir la conformité à diverses normes réglementaires.

D'autre part, les services hébergés stockent les données sensibles des clients hors site, ce qui peut susciter des inquiétudes quant à la confidentialité et à la fiabilité des données. Un environnement d'hébergement partagé peut présenter des risques supplémentaires en matière de sécurité. Les pannes ou les temps d'arrêt du fournisseur de services ont une incidence directe sur la disponibilité du service pour le client. Le fait de dépendre d'un seul fournisseur peut poser problème si celui-ci modifie ses conditions générales ou met fin au service.

Les services auto-hébergés conviennent donc aux organisations qui disposent de l'expertise technique et des ressources nécessaires pour gérer leur infrastructure. Cependant, l'auto-hébergement implique des coûts initiaux plus élevés, une maintenance continue et des défis en matière d'évolutivité. Le choix entre les services hébergés et auto-hébergés dépend en fin de compte des besoins, des capacités techniques, du budget et des priorités de chaque client en matière de contrôle, de personnalisation et de confidentialité des données.

Impact des licences

Fondamentalement, les licences de logiciels open source fonctionnent bien avec les modèles d'entreprises de fournisseurs de services, car le client ne s'attend pas à devenir propriétaire du logiciel qui fournit le service. Il paie plutôt pour avoir accès au service.

D'autre part, l'utilisation de logiciels libres dans des services tels que IaaS, SaaS ou PaaS soulève des questions juridiques qui ne sont souvent pas clarifiées. Seules quelques licences, notamment la *licence publique générale GNU Affero* (AGPL), réglementent explicitement l'utilisation des logiciels libres "dans le cas des logiciels serveurs réseau".

Avec d'autres licences populaires — qu'il s'agisse des licences copyleft plus restrictives telles que la licence publique générale GNU (GPL) ou des licences permissives telles que les licences BSD — il existe une marge d'appréciation quant à ce qui peut être considéré comme "copie", "transmission" ou "distribution" lorsque le logiciel est utilisé sur le réseau. Les décisions concernant la mise à disposition du code source à l'application ou les exigences en matière d'attribution dépendent de ces définitions. Le lien avec les composants logiciels propriétaires ou l'octroi de licences pour les composants supplémentaires développés en interne doit également être clarifié sur le plan juridique. Pour ces raisons, il est essentiel, dans le contexte des modèles commerciaux des fournisseurs de services, de clarifier les questions de licence liées à l'utilisation des logiciels libres.

Il est recommandé aux fournisseurs de services de publier la liste de tous les logiciels libres qu'ils utilisent, accompagnée de leurs licences respectives, même s'ils ne sont pas légalement tenus de le faire.

Considérations relatives à la sécurité et à la protection de la vie privée

Le succès du modèle d'entreprise d'un fournisseur de services repose en grande partie sur la satisfaction de ses clients, en particulier dans le domaine des logiciels open source, où les clients ont la liberté d'héberger eux-mêmes le logiciel ou de choisir un autre fournisseur de services pour le même logiciel. Ainsi, le principal avantage concurrentiel d'un fournisseur de services open source réside dans l'expérience client qu'il offre, qui peut être plus pratique, plus rentable, plus fiable, plus sûre ou plus performante que l'auto-hébergement. Cependant, le compromis est que les clients doivent partager leurs données avec le prestataire de services, ce qui peut soulever des questions relatives à la confidentialité des données personnelles ou sensibles.

Pour des raisons de sécurité, les clients doivent s'assurer que le fournisseur de services dispose d'un processus robuste permettant d'appliquer rapidement les correctifs de sécurité. Il est utile d'évaluer la manière dont le service gère les dépendances vis-à-vis d'autres projets open source afin de s'assurer que tous les logiciels sont à jour et sécurisés. Le client doit également évaluer les politiques de sécurité du fournisseur, notamment la manière dont il gère le chiffrement des données, les contrôles d'accès et la réponse aux incidents.

La transparence des logiciels open source permet à la communauté de procéder à une révision approfondie du code, ce qui peut renforcer la sécurité en identifiant et en corrigeant les vulnérabilités. Les clients doivent donc rechercher des évaluations ou des audits du logiciel réalisés par des tiers réputés ou des experts en sécurité au sein de la communauté. Ces audits doivent confirmer que le fournisseur respecte les pratiques et les normes de codage sécurisé pendant le processus de développement, et doivent vérifier si le fournisseur utilise des pipelines CI/CD avec des contrôles de sécurité intégrés pour détecter les vulnérabilités à un stade précoce.

Pour protéger leur vie privée, les clients doivent vérifier que le service ne collecte et ne stocke que les données nécessaires à son fonctionnement, afin de minimiser le risque d'exposition des données. Le service doit chiffrer les données transmises et stockées à l'aide d'un protocole de chiffrement puissant. Des mécanismes de contrôle d'accès doivent garantir que seul le personnel autorisé peut accéder aux données sensibles.

Le client doit vérifier que le service permet aux utilisateurs de contrôler leurs données, notamment en leur donnant la possibilité de les supprimer ou de les exporter. Le service doit respecter les réglementations applicables en matière de confidentialité, telles que le règlement

général sur la protection des données (RGPD) de l'Union européenne et la loi californienne sur la protection de la vie privée des consommateurs (*California's Consumer Privacy Act -CCPA*), et faire preuve de transparence quant à ses pratiques en matière de traitement des données.

Il est important de vérifier que le fournisseur dispose de politiques et de procédures claires pour répondre aux violations de données, y compris les exigences en matière de notification. Le client doit également examiner la politique de confidentialité du fournisseur afin de comprendre comment les données sont collectées, utilisées, partagées et protégées, et tenir compte des tiers avec lesquels le fournisseur peut partager des données et s'assurer qu'ils respectent également des normes de confidentialité strictes.

En général, il est utile de consulter les commentaires et les avis de la communauté pour évaluer la réputation et la fiabilité du logiciel et du fournisseur de services. Une communauté ou une organisation solide doit entretenir et soutenir activement le projet open source. Le client doit vérifier que le fournisseur dispose de procédures régulières de sauvegarde des données et de plans de reprise après sinistre solides, et vérifier si le service utilise la redondance des données pour garantir la disponibilité et la fiabilité.

Les fournisseurs de services doivent être conscients que les clients évalueront leurs pratiques en matière de sécurité et de confidentialité, leur réputation au sein de la communauté, leur conformité aux réglementations et leur transparence dans le traitement des données. Ils doivent donc prendre des mesures pour respecter les meilleures pratiques.

Accords communs entre le fournisseur de services et le client

Les fournisseurs de services adoptent généralement des conditions générales et des contrats qui constituent le fondement juridique et opérationnel de la relation entre l'entreprise et ses clients. Ces contrats permettent d'assurer une communication claire, de définir les attentes, de préciser les responsabilités et d'offrir une protection juridique aux deux parties.

Les *conditions générales d'utilisation* (CGU, ou ToS pour *Terms of Service*) d'un service, également appelées *conditions générales* (CG ou T&C pour *Terms and Conditions*) ou conditions d'utilisation, constituent un accord juridique entre un fournisseur de services et les clients ou utilisateurs de ce service. Les CGU définissent les règles, les responsabilités et les limitations qui régissent l'utilisation du service. Elles protègent à la fois le fournisseur de services et le client en définissant clairement ce que chaque partie peut et ne peut pas faire lors de la fourniture ou de l'utilisation du service. Parmi les éléments courants d'un accord CGU, on trouve la confirmation que l'utilisateur accepte de se conformer à ses conditions, des directives sur les comportements acceptables et les activités interdites, des détails sur la propriété du contenu créé ou téléchargé par les utilisateurs, des informations sur la création, la sécurité et la résiliation des comptes, les processus d'arbitrage ou de médiation pour résoudre les conflits, et les limites de la responsabilité

du fournisseur de services en cas de dommages.

Une *politique de confidentialité (Privacy Policy)* est une déclaration qui décrit comment le fournisseur de services collecte, utilise, stocke et protège les données des utilisateurs. Les éléments courants d'une politique de confidentialité sont les types de données collectées par le fournisseur de services et leurs méthodes de collecte, la manière dont ils utilisent les données, les conditions dans lesquelles le fournisseur de services peut partager les données avec des tiers, et les informations sur les droits des utilisateurs concernant leurs données, tels que l'accès, la correction et la suppression.

Un *accord de niveau de service (Service Level Agreement - SLA)* est un accord formel et documenté entre un fournisseur de services et un client qui définit le niveau de service attendu, y compris les indicateurs de performance spécifiques, les responsabilités et les attentes. Le SLA définit les normes de prestation de services, fournissant un cadre clair pour mesurer et gérer les performances des services. Parmi les éléments courants d'un SLA, on trouve une description détaillée des services fournis, des objectifs spécifiques en matière de performances des services tels que la disponibilité et les temps de réponse, des méthodes de mesure et de reporting des performances, les conséquences en cas de non-respect des objectifs de performance, et les procédures de révision et de mise à jour du SLA.

Un *objectif de niveau de service (Service Level Objective - SLO)* est un élément clé d'un accord de niveau de service qui spécifie des cibles ou des objectifs mesurables pour la performance du service. Ces objectifs quantifient le niveau de service attendu entre un fournisseur de services et un client et sont utilisés pour garantir que le service répond systématiquement aux normes convenues. Parmi les éléments courants des SLO, on trouve les indicateurs spécifiques que le fournisseur de services mesurera pour évaluer les performances du service (par exemple, le temps de disponibilité, le temps de réponse, le temps de résolution et le débit), les valeurs souhaitées ou attendues pour chaque indicateur de performance, les techniques et outils utilisés pour mesurer et surveiller les indicateurs de performance, les composants ou aspects spécifiques du service couverts par le SLO (par exemple, le matériel, les logiciels, les composants réseau ou des processus spécifiques), et les conséquences ou récompenses selon que le service respecte ou non les SLO (par exemple, pénalités financières, crédits de service ou primes de performance).

Un *accord sur le traitement des données (Data Processing Agreement - DPA)* est un contrat conclu entre un client (le propriétaire des données) et un fournisseur de services (le traitant des données) impliqué dans le traitement des données à caractère personnel. Le DPA détaille les responsabilités et les obligations des deux parties afin de garantir le respect des lois sur la protection des données, telles que le RGPD. Dans certaines juridictions, un DPA entre un fournisseur de services et un client est obligatoire.

Coûts et investissements

Les dépenses les plus importantes dans le modèle d'entreprise d'un fournisseur de services sont les coûts fixes liés à l'hébergement des services, tels que l'achat de matériel serveur, la location d'espace dans un centre de données, l'électricité nécessaire pour alimenter et refroidir les serveurs, la bande passante réseau et les salaires du personnel chargé du déploiement et de la maintenance. Certains fournisseurs de services peuvent également avoir des coûts variables liés aux licences logicielles ou aux services tiers. Les clients qui choisissent d'héberger eux-mêmes leurs services supportent directement ces coûts.

En choisissant des logiciels open source, les fournisseurs de services et les clients peuvent réduire ou éliminer les coûts de licence par rapport aux logiciels propriétaires, car, par définition, les logiciels open source ne facturent aucun frais pour la licence logicielle. L'utilisation de logiciels open source pour créer des services permet également de gagner du temps et d'économiser des ressources en matière de développement, tandis que la maintenance et les mises à jour assurées par la communauté peuvent réduire les coûts à long terme. Cependant, il est important de prendre en compte le coût total de possession (*Total Cost of Ownership* -TCO) des logiciels open source, y compris les coûts potentiels d'assistance, de maintenance et d'intégration.

Exercices guidés

1. Citez quelques moyens permettant aux fournisseurs de services de générer des revenus sans facturer les utilisateurs du service.

2. Quelles sont les raisons qui peuvent pousser un fournisseur de services à proposer son logiciel en auto-hébergement à ses clients ?

3. Pourquoi un fournisseur de services publierait-il son logiciel sous licence GNU Affero GPL ?

Exercices d'exploration

1. Qu'est-ce qu'une politique d'utilisation acceptable (PUA ou AUP pour *Acceptable Use Policy*) ?

2. Qu'est-ce qu'un contrat de licence utilisateur final (CLUF ou EULA pour *End User License Agreement*) ?

Résumé

Cette leçon traite des modèles d'entreprises des fournisseurs de services qui s'appuient sur des logiciels open source, ainsi que des différentes sources de revenus, notamment les abonnements, les recettes publicitaires, les tarifs horaires et les modèles basés sur des commissions. Elle met en évidence l'impact des licences de logiciels open source sur les fournisseurs de services, en soulignant leurs avantages et leurs obligations. Les considérations relatives à la sécurité, à la confidentialité et à la fiabilité des services, ainsi que l'importance des conditions générales d'utilisation (*Terms of Service* - ToS), des accords de niveau de service (*Service Level Agreement* - SLA) et des accords de traitement des données (*Data Processing Agreement* - DPA) sont également abordées.

Réponses aux exercices guidés

1. Citez quelques moyens permettant aux fournisseurs de services de générer des revenus sans facturer les utilisateurs du service.

Le service peut proposer des publicités, pour lesquelles les annonceurs paient. Le service peut facturer une commission aux fournisseurs qui proposent des produits ou des services sur le site, tels que la vente au détail ou les voyages. Le fournisseur de services peut vendre les données des clients, une pratique à laquelle de nombreux clients s'opposeraient et qui devrait être expliquée dans les conditions générales d'utilisation.

2. Quelles sont les raisons qui peuvent pousser un fournisseur de services à proposer son logiciel en auto-hébergement à ses clients ?

Les clients potentiels peuvent essayer le logiciel pour vérifier ses fonctionnalités et sa fiabilité avant de s'engager auprès du fournisseur de services. Les clients peuvent également trouver des bogues et même proposer des correctifs.

3. Pourquoi un fournisseur de services publierait-il son logiciel sous licence GNU Affero GPL ?

La licence Affero GPL exige des autres fournisseurs de services qu'ils publient toute modification apportée au logiciel sous la même licence. Cette disposition empêche les concurrents de tirer profit du fournisseur de services en ajoutant quelques améliorations qu'ils gardent propriétaires, puis en vantant leur version du service comme étant supérieure à l'originale.

Réponses aux exercices d'exploration

1. Qu'est-ce qu'une politique d'utilisation acceptable (PUA ou AUP pour *Acceptable Use Policy*) ?

Une politique d'utilisation acceptable (PUA) est une politique qui définit les utilisations acceptables et inacceptables du service, afin d'empêcher les abus et de garantir un environnement sûr et fiable. Parmi les éléments courants d'une PUA figurent une description des actions spécifiques qui ne sont pas autorisées, telles que le spam ou le piratage, les obligations des utilisateurs d'utiliser le service de manière responsable, les mesures que le fournisseur de services peut prendre en cas de violation, telles que la suspension du compte d'un utilisateur, et les procédures de signalement et de traitement des violations.

2. Qu'est-ce qu'un contrat de licence utilisateur final (CLUF ou EULA pour *End User License Agreement*) ?

Un contrat de licence utilisateur final (CLUF) est un contrat juridique conclu entre un fournisseur de logiciel propriétaire et un utilisateur final (et non une entreprise ou une organisation), qui accorde à l'utilisateur le droit d'utiliser le logiciel sous certaines conditions. Parmi les éléments courants des CLUF figurent la portée et les limitations de la licence (par exemple, utilisation personnelle ou licence non transférable), les actions interdites (par exemple, rétro-ingénierie ou redistribution), les conditions dans lesquelles la licence peut être résiliée, ainsi que la propriété et la protection des droits de propriété intellectuelle. Les services entièrement open source remplacent un contrat de licence d'utilisation propriétaire par une licence de logiciel open source.



054.3 Conformité et atténuation des risques

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 054.3

Valeur

3

Domaines de connaissance les plus importants

- Comprendre comment assurer la conformité des licences
- Comprendre comment conserver les informations sur les licences
- Comprendre le concept des bureaux de programme Open Source
- Comprendre les implications des droits d'auteur, des brevets et des marques déposées sur les modèles d'entreprise de l'open source
- Sensibilisation aux risques juridiques liés aux modèles d'entreprise des logiciels open source
- Sensibilisation aux risques financiers liés aux modèles d'entreprise des logiciels open source

Liste partielle de termes, fichiers et utilitaires utilisés

- Analyse de la composition du logiciel (SCA)
- Nomenclature des logiciels (SBOM)
- Échange de données sur les paquets de logiciels (SPDX)
- OWASP CycloneDX
- Bureaux de programme Open Source (OSPO)
- Garantie du produit
- Responsabilité du produit
- Réglementation sur l'export

- Impact des fusions et des acquisitions



Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	054 Modèles d'entreprises open source
Objectif :	054.3 Conformité et atténuation des risques
Leçon :	1 sur 1

Introduction

Une vieille maxime dans le milieu de l'open source dit que “les logiciels libres ne sont pas gratuits”. Si les logiciels libres et open source permettent une grande innovation et créativité, les utilisateurs et les développeurs doivent toutefois se conformer à un certain nombre de règles. Cette leçon traite de la conformité et des risques liés aux logiciels open source.

Cette leçon présente les étapes de base que les développeurs doivent suivre pour se conformer aux licences, les risques liés à l'utilisation de logiciels open source, ainsi que les méthodes permettant de suivre et de répertorier les logiciels utilisés par l'organisation. Nous verrons comment ces tâches peuvent être transformées en politiques et promues par un bureau des programmes open source (*Open Source Program Office* - OSPO).

Exigences relatives à la publication de logiciels basés sur des composants open source

Si l'organisation utilise des logiciels en interne, les licences de logiciels libres et open source n'imposent aucune exigence. L'organisation peut définir ses propres règles pour prévenir les

vulnérabilités, s'assurer que tout le monde utilise les mêmes composants ou pour d'autres raisons. Mais cela dépasse le cadre de cette leçon. Toutes les exigences imposées par les licences open source ou par l'organisation elle-même concernant l'utilisation des logiciels doivent être documentées, et l'organisation doit dispenser une formation sur ses politiques.

Même si l'organisation utilise des logiciels sur son serveur web et propose des services avec lesquels des personnes extérieures à l'organisation interagissent, la plupart des licences de logiciels libres et open source n'imposent aucune exigence. Cependant, la licence publique GNU Affero exige le respect des conditions du copyleft pour les logiciels fonctionnant également en tant que service.

Les exigences légales de chaque licence majeure de logiciel libre et open source ont été examinées en détail dans d'autres leçons. Cette section se contente donc de présenter une liste des mesures que les développeurs et les responsables doivent prendre pour s'y conformer :

Vérification des composants open source

L'organisation doit disposer de politiques claires et d'une formation sur les logiciels open source à utiliser et leur intégration dans les produits. Ces politiques couvrent la conformité, ainsi que d'autres questions telles que la sécurité et la participation à la communauté qui développe le logiciel.

Conserver la licence originale dans le code

Les développeurs ne doivent pas supprimer la licence du composant qu'ils utilisent. L'inclusion de la licence dans le code source est généralement une exigence de la licence. En fait, la suppression de la licence pourrait être considérée comme du plagiat, car le développeur donne l'impression d'avoir écrit le code. La suppression de la licence pourrait également entraîner de graves problèmes par la suite, parce que l'organisation pourrait enfreindre la licence lorsque le code est inclus dans ses produits.

Surveiller les licences

L'organisation doit tenir à jour un catalogue répertoriant les licences de chaque fichier ou paquet utilisé par l'organisation, afin de s'assurer qu'elle respecte les licences.

Reconnaissance de l'auteur

Presque toutes les licences exigent que le propriétaire des droits d'auteur (souvent appelé "auteur") soit mentionné lors de la discussion et de la promotion du logiciel. Chaque licence explique ce qui doit être inclus dans cette mention : en général, elles obligent une mention de copyright standard avec l'année et le nom du propriétaire des droits d'auteur. Cette mention doit apparaître dans toute documentation relative au produit utilisant le composant, y compris la publicité et le site web de l'appareil ou du programme.

Ne pas suggérer une approbation

Certaines licences BSD exigent que la personne qui distribue un produit contenant ces composants veille à ne pas suggérer que le détenteur des droits d'auteur approuve le produit. Même si cette exigence n'est pas explicitement mentionnée, il est éthique de s'y conformer.

Offrir le code source pour les licences copyleft

Si une organisation distribue un fichier binaire contenant des composants sous licence copyleft, que ce soit sous forme de distribution logicielle ou sur un appareil, elle doit mettre à la disposition du public le code source de ces composants. Si l'organisation modifie le code et redistribue l'œuvre dérivée sous forme binaire, le code source de la version modifiée (*dérivée*) doit également être mis à la disposition du public. La distribution peut se faire par tout moyen facilitant l'accès au public, par exemple en publiant le code sur un site web ou en le proposant sur un disque ou une clé USB.

Ne pas inclure de composants sous licence copyleft dans des produits propriétaires

Si l'organisation intègre des composants sous licence copyleft dans un produit, elle peut, dans certaines circonstances, être tenue de publier l'intégralité du produit sous la même licence copyleft. Les conditions qui déclenchent cette obligation varient d'une licence à l'autre. Il est toutefois parfois difficile de déterminer quel type d'utilisation déclenche l'obligation de copyleft. Ainsi, sauf dans des situations claires telles que l'utilisation d'une bibliothèque open source (qui ne devrait pas déclencher l'exigence réciproque du copyleft), les développeurs doivent être très prudents quant à leur utilisation de composants sous copyleft, à moins que l'organisation ne soit prête à publier son produit sous la même licence.

Documenter les modifications apportées au code

Lorsque vous distribuez des versions modifiées du code, indiquez clairement les modifications apportées par l'organisation.

Accorder le droit de brevet

Certaines licences de logiciels libres ou open source exigent l'octroi d'un brevet pour l'utilisation du logiciel. Ainsi, si une organisation diffuse du code sous une telle licence et détient un brevet sur un processus du code, l'organisation ne peut pas faire payer une redevance ou exercer d'autres contrôles basés sur son brevet à quiconque utilise ou adapte le code.

Respecter les marques déposées

Certains logiciels libres sont couverts par des marques déposées. Les marques déposées peuvent couvrir des mots et des phrases, des logos et d'autres images, et bien d'autres choses encore. D'une part, une organisation doit gérer correctement la marque déposée pour tout logiciel qu'elle utilise : Une violation courante consiste à parodier, modifier ou simplement

afficher une marque déposée sans autorisation. D'autre part, si l'organisation souhaite utiliser la marque déposée, elle doit se conformer aux exigences du propriétaire de la marque déposée, ce qui peut exclure toute modification du logiciel.

Risques liés aux logiciels open source

Cette leçon porte sur la conformité, nous nous concentrerons donc sur les risques dans ce domaine, mais nous aborderons également d'autres sujets.

Violation de licence

Les licences de logiciels libres et open source doivent être traitées avec le même sérieux que les autres licences et contrats. Des organisations les font respecter, comme le Software Freedom Conservancy, qui agit au nom des petits projets protégés par le copyleft qui ne disposent pas de leurs propres mécanismes d'application.

Ces organisations n'envisagent généralement les poursuites judiciaires qu'en dernier recours. La plupart des violations ne sont pas intentionnelles et sont faciles à résoudre par l'éducation. Néanmoins, il est dommageable pour la réputation d'une entreprise d'être perçue comme ignorante et insensible à l'égard des communautés dont elle dépend pour ses logiciels.

Des poursuites ont été engagées lorsqu'un utilisateur de logiciel refusait de manière flagrante de coopérer et lorsque le logiciel et le défendeur étaient suffisamment importants.

Même si une entreprise n'est pas poursuivie en justice, les dommages causés par une violation de licence à ses relations avec la communauté, ainsi qu'à sa réputation, peuvent être considérables. Un projet peut dérailler pour une raison aussi simple que le fait que les questions des développeurs restent sans réponse sur les forums consacrés au logiciel qu'ils essaient d'utiliser.

Il peut être désastreux pour quelqu'un de trouver un logiciel sous copyleft dans un produit propriétaire. Pour se conformer à la licence, les développeurs doivent soit supprimer tout le code protégé par le copyleft, soit libérer leur produit sous la licence copyleft. Le fait de rendre leur logiciel libre, avec le code source disponible, pourrait avoir des effets désastreux sur les modèles d'entreprise basés sur les droits de licence ou d'autres moyens de monopoliser la valeur du produit.

Confiance et réputation

Nous avons vu que les violations de licence peuvent être très préjudiciables. D'autres types de comportement qui perturbent la confiance et la réputation présentent également des risques.

Certaines organisations publient des logiciels sous une licence libre ou open source, puis annoncent à un moment donné que les versions futures seront soumises à une licence propriétaire. Ce changement peut être tentant, car de nombreux projets de logiciels libres ne bénéficient pas du soutien financier d'un nombre suffisant de clients.

Mais ces changements de licence provoquent la colère des clients et des développeurs externes qui contribuent au projet. Parfois, les développeurs externes s'emparent de la version libre et continuent à la développer de manière indépendante, ce que l'on appelle le *forking* du projet. La version libre peut devenir concurrentielle par rapport à la version propriétaire de l'entreprise et détourner les clients de cette dernière.

La participation à des forums de projets comporte également des risques. Une entreprise doit apprendre à être un bon citoyen. Les problèmes les plus fréquents sont les suivants :

- Essayer d'utiliser les contributions financières ou en code d'une entreprise comme levier pour forcer le projet à prendre une direction que les autres développeurs ne souhaitent pas
- Exiger trop de la communauté, par exemple en la soumettant à de nombreuses demandes de fonctionnalités ou même à un trop grand nombre de questions
- Être malpoli d'une autre manière et violer le code de conduite du projet
- Essayer de monopoliser la publicité ou de tirer profit de la réussite du projet de manière inappropriée et sous d'autres formes

Investissements sans contrepartie

Les modèles d'entreprise sont abordés dans une autre leçon ; la présente section ne fait que mentionner le risque financier lié à la participation à des projets de logiciels open source.

Les entreprises peuvent lancer des projets open source ou y participer dans l'espoir d'en tirer des revenus par le biais d'un mécanisme quelconque, tel que des contrats d'assistance, des contrats SaaS, la collecte de données, voire des dons et des subventions. Malheureusement, ceux qui mènent des projets open source les trouvent souvent moins lucratifs qu'ils ne l'espéraient. Bien sûr, toute entreprise prend un risque en lançant un nouveau projet, mais il est particulièrement difficile de trouver une source de revenus fiable pour l'open source.

Les logiciels open source semblent plus durables lorsqu'ils soutiennent une autre forme de revenu, comme la vente de périphériques, de voitures, d'imprimantes, etc.

Forks

Comme nous l'avons vu, les membres d'un projet sont parfois en désaccord sur la feuille de route,

le leadership ou d'autres aspects du projet et créent un projet alternatif, un fork, sur la même base de code. La création du fork peut également être le fait d'une organisation qui souhaite créer une version spécialisée d'un logiciel. Par exemple, Android utilise une version spécialisée de Linux que Google maintient. (Google apporte aussi de nombreuses contributions à la version principale de Linux, qui ne sont pas toujours acceptées).

Les organisations peuvent créer un fork pour diverses raisons. Elles peuvent soumettre leurs modifications au projet principal et les voir rejetées, parce que d'autres développeurs n'en veulent pas. Dans le cas d'un projet sous licence permissive, ou d'un projet utilisé uniquement au sein de l'organisation, celle-ci peut vouloir garder ses modifications secrètes.

Le risque du fork est que les développeurs du projet alternatif sont responsables de la maintenance de l'ensemble du produit. Si d'importantes corrections de bogues ou de nouvelles fonctionnalités sont ajoutées au projet original, les développeurs du projet alternatif doivent reproduire les changements ou renoncer à leurs avantages. Au fur et à mesure que le temps passe et que les projets s'éloignent les uns des autres, il devient de plus en plus difficile de suivre l'évolution du projet original.

Licences incompatibles

Comme expliqué dans d'autres leçons, certaines licences de logiciels libres et open source comportent des clauses incompatibles. De tels composants ne peuvent pas être utilisés ensemble dans un produit.

Ce problème est susceptible de se poser lors d'une fusion ou d'une acquisition. Chaque entreprise peut avoir utilisé un logiciel avec une licence particulière, et si elles veulent combiner le code dans leurs projets, elles doivent s'assurer que les licences sont compatibles.

Responsabilité du fait du produit

De nombreuses licences de logiciels open source (et les conditions générales d'utilisation de nombreux logiciels et services propriétaires, d'ailleurs) déclinent explicitement toute responsabilité quant à l'utilisation du logiciel. On peut également dire que la licence ou les conditions générales d'utilisation n'offrent aucune garantie.

Les vendeurs de logiciels sont rarement tenus pour responsables des problèmes liés à leurs logiciels, mais les clients les poursuivent parfois en justice ou essaient d'autres formes de sanction. Les tribunaux ne sont pas obligés de reconnaître les clauses de "non-garantie".

De plus en plus, les lois et les règlements imposent des responsabilités aux créateurs de logiciels. Ces restrictions légales — ou du moins les propositions de restrictions — sont particulièrement visibles aujourd'hui dans le domaine de l'intelligence artificielle, mais elles ont parfois une portée

plus large.

Règlements sur l'exportation

De nombreux pays limitent l'exportation de biens et de logiciels. Dans le domaine des logiciels, ces réglementations s'appliquent le plus souvent aux produits de sécurité, et plus particulièrement à l'utilisation du chiffrement. Les États-Unis avaient l'habitude d'appliquer des contrôles stricts à tout logiciel contenant du chiffrement, mais ces contrôles ont été assouplis pour les projets open source.

Les entreprises doivent connaître les réglementations en matière d'exportation en vigueur dans les pays où elles créent des logiciels, au cas où ces réglementations affecteraient la vente et la distribution de leurs produits.

Nomenclature de logiciels : Savoir ce que vous utilisez

De nombreux produits sont accompagnés d'une nomenclature, qui n'est autre qu'une liste de tous leurs composants. Par exemple, lorsque vous achetez un produit de consommation, celui-ci peut être accompagné d'une feuille de papier énumérant toutes les pièces, jusqu'aux écrous et aux boulons. La liste peut contenir des informations utiles telles que les dimensions d'une pièce et un numéro de pièce qui vous permettra de commander une nouvelle pièce.

Les projets open source incluent désormais une *nomenclature de logiciels* (*Software Bill of Materials* - SBOM, prononcé "S-bomb"). Au minimum, un SBOM énumère les paquets, les noms de fichiers, les licences, les auteurs ou propriétaires et les numéros de version. De nombreux SBOM vont plus loin et incluent des informations sur les failles de sécurité.

Les projets génèrent un SBOM pour chaque version à l'aide d'outils automatisés. Les utilisateurs peuvent parcourir le SBOM pour trouver les informations dont ils ont besoin. Par exemple, l'extraction des licences aide l'organisation à décider rapidement si les composants sont compatibles avec leur propre code et s'il est légalement permis d'utiliser les composants avec leur code.

De même, l'extraction d'informations sur la version de chaque paquet permet à l'organisation de découvrir si les différentes parties de son produit dépendent de différentes versions d'une bibliothèque particulière. L'utilisation de deux versions de la bibliothèque entraîne au minimum un gonflement. Elle peut également être source de confusion et d'erreurs si un composant est construit avec la mauvaise version.

Normes pour les SBOM

Étant donné que les environnements informatiques utilisent d'énormes quantités (parfois des dizaines de milliers) de composants et qu'ils font l'objet de changements fréquents, les SBOM doivent être hautement structurés afin que les informations puissent être extraites automatiquement et rapidement. Cette section décrit deux des formats les plus populaires dans le monde de l'open source :

Software Package Data Exchange (SPDX)

Le format représente chaque paquet et tout le contenu de ce paquet dans une structure arborescente. Le format documente les dépendances entre les fichiers et d'autres relations. Des pointeurs vers des informations, connus sous le nom de "snippets", peuvent être créés et utilisés dans tout le document de manière que les informations ne soient définies qu'à un seul endroit. Ce format a été développé par la Fondation Linux.

CycloneDX

Il s'agit d'un format plus large avec des champs plus détaillés, en particulier pour les informations sur les vulnérabilités. Ce format a été créé par le *Open Worldwide Application Security Project* (OWASP) et est populaire dans le secteur militaire et dans d'autres organisations qui mettent l'accent sur la sécurité. Par exemple, l'entrée d'un fichier peut inclure le nom d'une vulnérabilité de sécurité, la source d'information sur la vulnérabilité, les cibles affectées, etc. Ce format est également conçu pour les déploiements en cloud qui peuvent inclure des milliers de systèmes différents.

SPDX et CycloneDX créent tous deux des structures hiérarchiques dans différents formats, notamment JSON et XML. De nombreux outils existent pour chaque norme, à la fois pour créer les formats et pour analyser les fichiers créés à la recherche d'informations. Les sites de dépôt de contrôle de version les plus populaires permettent aux développeurs de créer un SBOM dans l'un de ces formats en cliquant sur un bouton.

Analyse de la composition des logiciels

Pour savoir ce qu'une organisation utilise, il faut évaluer ses logiciels et comprendre leur origine, les vulnérabilités qu'ils contiennent, les licences qu'ils utilisent et d'autres facteurs qui aident une personne ou une organisation à décider d'utiliser ou non le logiciel. Cette tâche est appelée *analyse de la composition des logiciels* (*Software Composition Analysis* - SCA), et il existe de nombreux outils permettant d'effectuer des analyses sophistiquées des SBOM et des logiciels eux-mêmes.

Certains outils récupèrent des informations sur les licences, ce qui aide une organisation à décider si le logiciel peut être inclus en toute sécurité dans un produit et si les différents composants ont

des licences compatibles. Certains outils comparent même des extraits de code à des bibliothèques de projets open source courants afin de déterminer si le code a été repris des projets open source sans attribution appropriée.

L'utilisation de ces outils est particulièrement cruciale lors d'une fusion ou d'une acquisition. L'entreprise acquéreuse peut se rendre compte que le logiciel qu'elle souhaite acquérir a moins de valeur que prévu, car il contient des composants open source qui imposent des exigences interférant avec l'utilisation prévue dans la nouvelle organisation.

Stratégies formelles et conformité

Les processus et les techniques décrits dans cette leçon doivent être élaborés par les responsables avec l'aide des développeurs, des avocats et d'autres personnes compétentes. Cette section décrit certaines des décisions stratégiques que les organisations doivent prendre concernant les logiciels open source. Nous terminerons par une description d'un *Open Source Program Office* (OSPO), qui peut être un défenseur et une source d'information pour les stratégies.

Principes régissant l'utilisation et la contribution des logiciels open source

Les organisations peuvent tirer un grand profit de l'utilisation et de la contribution aux logiciels open source, mais elles doivent le faire de manière à protéger leurs propres intérêts tout en profitant aux projets open source. Des stratégies doivent être définies pour :

- Où utiliser les logiciels open source (opérations, projets internes, produits destinés aux clients, etc.)
- Ce qui fait qu'un projet open source convient à l'organisation : caractéristiques, performances, sécurité, feuille de route pour les extensions futures et viabilité de la communauté
- Analyse de la composition des logiciels et de l'endroit où la documentation qui en résulte doit être stockée
- Récompenses pour les développeurs contribuant aux logiciels open source, y compris leur participation à des forums publics
- Lignes directrices pour la contribution aux projets, y compris la manière de représenter publiquement l'entreprise
- Les exigences en matière de documentation, afin que les développeurs en dehors de l'équipe principale puissent comprendre comment contribuer et interagir

L'OSPO peut coordonner l'élaboration de ces stratégies et la formation afin d'en assurer le respect.

Accords de contribution

Les projets open source doivent s'assurer que les contributions sont légitimes : par exemple, que le code n'a pas été repris d'un produit propriétaire ou d'un projet open source dont la licence est incompatible. De nombreux projets open source exigent des développeurs qu'ils fournissent un document appelé *Contributor License Agreement* (CLA) pour garantir la légitimité de leur contribution.

Les développeurs d'une organisation qui contribuent à ces projets peuvent demander aux juristes de l'organisation d'examiner et d'approuver la CLA. Les juristes doivent donc comprendre la pertinence des clauses des CLA et être prêts à les vérifier.

De nombreuses critiques ont été formulées à l'encontre des CLA, notamment leur complexité et les failles qu'elles laissent aux projets pour utiliser le code des contributeurs d'une manière non souhaitée par ces derniers.

De nombreux projets, au lieu d'une CLA, demandent au développeur de signer un court document appelé *Developer Certificate of Origin* (DCO), attestant que le développeur a le droit de contribuer au code. Le DCO, qui fait l'objet d'une autre leçon, n'est généralement pas soumis à l'examen d'un avocat.

Certains projets demandent simplement aux contributeurs de céder les droits d'auteur de leur code au projet dans un *Copyright Assignment Agreement* (CAA). Cependant, de nombreux contributeurs n'apprécient pas cette pratique, parce qu'ils ne peuvent pas utiliser le code dans leur propre projet et parce qu'elle confère beaucoup de pouvoir au projet.

Open Source Program Office (OSPO)

Les projets open source combinent des facteurs sociaux, techniques, juridiques et organisationnels inhabituels. À l'heure actuelle, la connaissance de ces facteurs ne s'est pas répandue dans les organisations au point que tous les gestionnaires, développeurs, avocats et autres les comprennent en profondeur. Les organisations peuvent donc tirer un grand profit de la création d'un *Open Source Program Office* (OSPO) comme point central pour la défense des intérêts, l'élaboration de stratégies, le renforcement des stratégies et la formation.

En tant que service polyvalent, les OSPO peuvent jouer de nombreux rôles, tels que :

Promouvoir l'open source

Les OSPO peuvent diffuser les concepts qui sous-tendent le mouvement open source et les suggestions d'utilisation de l'open source au sein de leur organisation. Ils peuvent rappeler aux développeurs de rechercher des solutions open source aux problèmes qu'ils résolvent et les encourager à adopter les logiciels appropriés. Ils peuvent inciter les responsables à accorder du

temps aux développeurs pour qu'ils participent à des projets open source et à reconnaître cette participation lors de l'évaluation des développeurs en vue d'une augmentation de salaire ou d'une promotion.

Définition des stratégies

Les OSPO peuvent mobiliser les gestionnaires pour qu'ils élaborent des stratégies et mettent en place des référentiels à cet effet.

Renforcement de la stratégie

Les OSPO peuvent rappeler aux développeurs de scanner les logiciels et les SBOM et de suivre les règles de l'entreprise concernant l'utilisation des logiciels open source. Les OSPO peuvent conserver la documentation sur les logiciels adoptés et d'autres aspects de l'utilisation par l'entreprise.

Formation

Les OSPO peuvent expliquer aux développeurs comment évaluer les projets de logiciels open source et y participer, expliquer aux juristes les détails des licences et autres considérations juridiques, et aider l'entreprise à comprendre les changements organisationnels et culturels qui facilitent l'utilisation des logiciels open source.

Il existe de nombreux documents et ressources en ligne pour créer une OSPO. Une petite organisation peut confier cette tâche à un tiers expert dans ce domaine.

Exercices guidés

1. Pourquoi un développeur ne pourrait-il pas simplement prendre le code qu'il aime dans un projet open source et l'intégrer dans son propre code sans préserver la licence ?

2. Quels types de documents un développeur doit-il signer lorsqu'il contribue à un projet open source ?

3. Vous trouvez un logiciel libre qui pourrait constituer une excellente base pour votre site de vente au détail. Pouvez-vous superposer votre logo à son logo populaire afin de créer une image accrocheuse pour votre site web ?

Exercices d'exploration

1. Quelles considérations pourraient vous amener à créer un fork d'un projet open source pour votre propre usage, malgré les inconvénients des forks ?

2. Lorsque vous trouvez un logiciel libre qui répond à vos besoins, quelles sont les raisons pour lesquelles vous pourriez le rejeter ?

3. Vous aimeriez utiliser un outil de journalisation sous licence copyleft dans votre produit propriétaire. Existe-t-il un moyen de les combiner sans avoir à proposer votre produit sous une licence copyleft ?

Résumé

Cette leçon a abordé de nombreux aspects à prendre en compte avant d'utiliser des logiciels libres et open source. Elle a expliqué comment respecter les licences, les différents risques juridiques, financiers et liés à la réputation, ainsi que la manière de définir des stratégies importantes et de les appliquer au sein de votre organisation.

Réponses aux exercices guidés

1. Pourquoi un développeur ne pourrait-il pas simplement prendre le code qu'il aime dans un projet open source et l'intégrer dans son propre code sans préserver la licence ?

Il s'agit généralement d'une violation de la licence open source, ainsi que d'un plagiat et d'une violation du droit d'auteur. En pratique, l'organisation peut être contrainte de se conformer à la licence, avec des conséquences allant de l'embarras et de l'atteinte à la réputation à la destruction de son modèle d'entreprise si le code protégé par le copyleft est mélangé au code propriétaire.

2. Quels types de documents un développeur doit-il signer lorsqu'il contribue à un projet open source ?

Un *Contributor License Agreement* est un document juridique qui accorde à l'organisation open source les droits d'utilisation du code. Un *Developer Certificate of Origin* est un document plus court et plus simple promettant que le développeur a le droit de contribuer au code. Un *Copyright Assignment Agreement* accorde tous les droits à l'organisation destinataire.

3. Vous trouvez un logiciel libre qui pourrait constituer une excellente base pour votre site de vente au détail. Pouvez-vous superposer votre logo à son logo populaire afin de créer une image accrocheuse pour votre site web ?

Si le projet a protégé son logo par une marque déposée, votre modification constituerait probablement une violation de cette marque déposée. Même si le logo n'est pas protégé par une marque déposée, votre modification pourrait être perçue comme un manque de respect et une source de confusion.

Réponses aux exercices d'exploration

1. Quelles considérations pourraient vous amener à créer un fork d'un projet open source pour votre propre usage, malgré les inconvénients des forks ?

Si vous êtes satisfait de l'état actuel du code, vous n'aurez peut-être pas besoin de suivre les modifications apportées au projet original. Il se peut que vous souhaitiez créer un produit substantiellement différent de l'usage qui est fait du projet original, et que vous soyez donc prêt à vous séparer de ce projet. Le code peut avoir suffisamment de valeur dans votre plan d'entreprise pour que votre équipe soit prête à assumer l'entière responsabilité de son développement et de sa maintenance.

2. Lorsque vous trouvez un logiciel libre qui répond à vos besoins, quelles sont les raisons pour lesquelles vous pourriez le rejeter ?

Le code peut contenir trop de bogues et de failles de sécurité, la communauté des développeurs du projet peut ne pas bien fonctionner, vous pouvez ne pas aimer la direction dans laquelle le code évolue, ou la licence peut ne pas être compatible avec d'autres codes de votre produit.

3. Vous aimeriez utiliser un outil de journalisation sous licence copyleft dans votre produit propriétaire. Existe-t-il un moyen de les combiner sans avoir à proposer votre produit sous une licence copyleft ?

L'intégration du code de l'outil dans votre code peut déclencher l'exigence de réciprocité du copyleft, en fonction de la licence en vigueur. L'outil de journalisation doit être séparé de votre produit afin que celui-ci ne soit pas considéré comme dérivé. Par exemple, il peut être prudent d'exécuter l'outil sous copyleft en tant que processus distinct et de communiquer avec lui par le biais d'envois de messages.



Thème 055: Gestion de projets



055.1 Modèles de développement de logiciels

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 055.1

Valeur

3

Domaines de connaissance les plus importants

- Comprendre la pertinence et les objectifs de la gestion de projet dans le développement de logiciels
- Comprendre les bases du développement de logiciels en cascade
- Comprendre les bases du développement agile de logiciels, y compris Scrum et Kanban
- Comprendre le concept de DevOps

Liste partielle de termes, fichiers et utilitaires utilisés

- Phases des projets en cascade (ingénierie des exigences, analyse commerciale, conception de logiciels, développement, tests, opérations)
- Rôles dans les projets en cascade (chefs de projet, analystes commerciaux, architectes logiciels, développeurs, testeurs)
- Organisation de projets Scrum (sprints et planification de sprint, carnet du produit et de sprint, mêlée quotidienne, revue de sprint et rétrospective de sprint)
- Rôles dans les projets Scrum (propriétaires de produits, développeurs, chefs de mêlée)
- Organisation de projets Kanban (tableaux Kanban)



Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	055 Gestion de projets
Objectif :	055.1 Modèles de développement de logiciels
Leçon :	1 sur 1

Introduction

La vie est pleine de projets. Il peut s'agir de planifier une soirée cinéma, un voyage ou un événement familial, de programmer la semaine des enfants ou d'améliorer votre travail dans le cadre d'un hobby. Peu importe ce que vous voulez réaliser : Vous devez faire des plans et prendre des mesures. Nous avons l'habitude de créer plusieurs scénarios, avec un plan B, un plan C, etc. Il en va de même dans le monde du développement de logiciels.

Rôles dans le développement de logiciels

La gestion d'un projet nécessite des compétences variées. Il est donc utile de faire appel à des personnes différentes pour remplir des rôles bien définis. Les sections suivantes décrivent certains des rôles que l'on retrouve couramment dans les projets de développement de logiciels.

Chef de projet

La gestion d'un projet est une tâche extrêmement complexe. Certaines situations semblent faciles à première vue, mais il faut du soin et de l'expérience pour y parvenir. La personne chargée de cette tâche est le *chef de projet* (*project manager*).

Les responsabilités du chef de projet consistent notamment à veiller à ce que le projet soit achevé dans les délais, dans le respect du budget et avec une qualité acceptable. Même s'ils n'écrivent pas une seule ligne de code, les chefs de projet sont responsables des résultats de l'équipe. Ils doivent s'aligner sur les clients en ce qui concerne les délais. Ils discutent également avec les personnes occupant d'autres fonctions pour s'assurer que tout est en ordre.

Analyste commercial et ingénieur des exigences

La micro-gestion n'a pas sa place—du moins dans un environnement de travail sain. Les *analystes commerciaux* (*Business analysts* - BA) et les *ingénieurs des exigences* (*requirement engineers* - RE), internes ou externes à l'équipe de projet, sont les bras droits des chefs de projet. Ils sont chargés de créer une documentation riche et claire sur les exigences. Ils sont également le pont entre les clients et les développeurs. Les clients communiquent en langage clair, et le BA/RE s'assure que la documentation est claire et sans équivoque, ce qui permet aux développeurs de travailler efficacement.

Imaginez que l'on vous demande d'apporter un “gros gâteau” à une fête. Que signifie “gros” ? Dix tranches ou vingt ? Il s'agit peut-être du gâteau d'un grand mariage et il devrait contenir cent parts. Le BA/RE doit spécifier exactement les exigences telles que les quantités, afin que les développeurs sachent, par exemple, qu'une application est conçue pour un nombre spécifique d'utilisateurs.

Nous n'en sommes qu'au début de la définition des exigences, mais vous pouvez déjà constater l'importance d'une communication claire. La communication est essentielle pour tout travail en commun, mais elle est peut-être encore plus importante dans le cadre d'un projet open source, car des personnes ayant différents antécédents peuvent travailler sur chaque partie du projet. Quelqu'un peut collaborer en tant qu'étudiant, quelqu'un d'autre en tant que parent, une personne à la retraite ou entre deux emplois, etc. Malgré cela, les progrès doivent se faire en douceur dans cet environnement—la communication ne peut donc pas freiner la progression.

Développeurs, architectes et testeurs

Pour implémenter les exigences, un projet de logiciel a besoin de *développeurs* qui créent le produit sur la base des documentations et des décisions de conception. Leur travail est jugé par l'*architecte*, qui possède généralement les connaissances techniques et spécialisées les plus étendues de toutes les personnes impliquées dans le projet. Chaque développeur, en particulier les plus expérimentés, est responsable de son propre code, mais les grandes décisions sont prises ou approuvées par l'architecte.

Parfois, des personnes sont spécifiquement désignées comme *testeurs*, qui sont responsables de toutes sortes de tests, de la documentation des résultats et de la création de rapports de

problèmes.

Planification et ordonnancement

Maintenant que nous avons abordé les rôles de base, parlons des phases d'un projet : planification, ordonnancement, implémentation, maintenance/test et livraison. Certaines phases diffèrent selon les modèles de développement de logiciels, mais la planification et l'ordonnancement sont des sujets généraux que nous aborderons avant de nous plonger plus profondément dans les différents modèles.

La *planification* a lieu après que les BA/RE ont fourni la liste des exigences. La planification consiste en une ou plusieurs réunions sur ce que l'équipe veut réaliser, comment elle prévoit de le faire et combien de temps cela prendra. En général, une analyse des risques est également effectuée lors de ces réunions.

Les planificateurs doivent ensuite *ordonnancer* le travail et créer des *étapes (milestones)*. À chaque étape, l'équipe sait qu'une partie importante est terminée. Les composants ou fonctionnalités à long terme peuvent prendre des mois, de sorte que les planificateurs doivent tenir compte des vacances, des jours fériés et — surtout pendant les saisons froides — des congés de maladie, afin de garantir un délai de livraison précis et d'éviter la précipitation et la panique à l'approche de la date butoir. Avec un bon calendrier, les développeurs se sentent plus détendus et peuvent fournir des solutions de haute qualité dans les délais impartis, sans défaillances.

Outils courants

Deux outils utiles à tous les modèles, et à la gestion de projet en général, sont un *système de contrôle de version (version control system - VCS)* et une *plateforme de gestion du cycle de vie des applications (application lifecycle management - ALM)*. Le contrôle des versions est abordé en profondeur dans une autre leçon, alors parlons un peu des ALM.

L'ALM est un cadre complet qui gère le cycle de vie d'une application logicielle, depuis le développement initial jusqu'à la maintenance et au retrait éventuel. L'ALM intègre les personnes, les processus et les outils pour améliorer la collaboration et l'efficacité, en veillant à ce que toutes les étapes du cycle de vie du logiciel soient bien coordonnées et alignées sur les objectifs de l'entreprise. Cette approche permet de maintenir la qualité, les performances et la fiabilité des applications tout au long de leur durée de vie.

Maintenant que nous avons une vue d'ensemble des rôles et des phases du cycle de vie, nous allons nous pencher sur les modèles.

Modèle en cascade

Il s'agit de l'une des méthodologies les plus anciennes et les plus simples en matière de développement de logiciels. On peut l'imaginer comme un escalier, où chaque marche doit être terminée avant d'avancer. Mais le modèle ne va que dans une seule direction, ce qui le rend adapté aux projets dont les exigences sont claires et les changements attendus minimes, voire inexistantes.

Bien que la simplicité de ce modèle puisse être un avantage, la rigidité de la méthodologie peut être un inconvénient lorsque la flexibilité et l'adaptabilité sont nécessaires. Et de nos jours, généralement, tout change pendant que vous travaillez sur votre projet.

Comme l'expliquent les sections précédentes, pour commencer le développement, un projet doit disposer des exigences, d'un plan fini et d'un ordonnancement définitif du travail avec des étapes. Dans le modèle en cascade, ces éléments sont le résultat de l'ingénierie des exigences et de l'analyse commerciale.

Ingénierie des exigences

Au cours de cette phase initiale, toutes les exigences du projet sont rassemblées et documentées. Ce travail implique des discussions approfondies entre le chef de projet et les parties prenantes afin de comprendre leurs besoins et leurs attentes. Le résultat est un document complet de spécification des exigences qui décrit ce que le système doit faire.

Analyse commerciale

Les analystes commerciaux examinent les exigences d'un point de vue commercial afin de s'assurer qu'elles correspondent aux objectifs de l'organisation. Ils réalisent des études de faisabilité, des évaluations des risques et des analyses coûts-avantages pour déterminer si le projet est viable et s'il en vaut la peine. Les connaissances acquises sont utilisées pour affiner la portée et les objectifs du projet.

Conception de logiciels

La conception de logiciels est l'étape au cours de laquelle l'architecte crée une spécification de conception détaillée que les développeurs doivent suivre. À partir de cette spécification, l'équipe peut commencer à implémenter les exigences — en d'autres termes, démarrer la phase de développement, qui vient ensuite.

Développement

La phase de développement est celle où la magie opère — celle où le code est écrit. En suivant assidûment la documentation et les décisions de conception, les développeurs écrivent leurs parties. Une fois que les développeurs ont revu leur code avec d'autres développeurs ou avec l'architecte, cette phase peut être considérée comme terminée.

Tests

Le développement est suivi d'une phase de test, gérée par les testeurs. Il existe différents types de tests, décrits dans une autre leçon, tels que les tests unitaires, les tests d'intégration, les tests système et les tests d'acceptation.

L'objectif de ces tests est de faire remonter les problèmes à la surface avant la mise en production et de permettre aux développeurs de les corriger à temps — et non pas une fois que le projet est déployé et publié et que les utilisateurs peuvent être irrités et frustrés par les bogues.

Mise en service

Une fois les tests passés et les bogues corrigés, le projet peut être livré, c'est-à-dire mis en place dans l'environnement de production. Cette phase peut comprendre l'installation, la configuration et même parfois la formation des utilisateurs. Une fois que le projet est prêt à être utilisé, il se trouve dans une phase de maintenance, au cours de laquelle l'équipe peut traiter les problèmes rencontrés par les utilisateurs et fournir des mises à jour si nécessaire.

Évaluation du modèle en cascade

Qu'avez-vous remarqué en lisant sur ce modèle ? Le trouvez-vous efficace ? Y a-t-il quelque chose qui manque ? Voyons les avantages et les inconvénients.

Le modèle en cascade présente les avantages suivants :

Structure claire

Le processus linéaire facilite la gestion, la compréhension et le suivi.

Documentation minutieuse

Une documentation détaillée et complète à chaque phase aide l'équipe à comprendre ce qui est nécessaire pendant le développement et la maintenance future.

Prévisibilité

Des étapes claires et bien définies permettent d'établir un calendrier et un budget prévisibles.

Gestion de projet simplifiée

Grâce à son déroulement linéaire et direct, le modèle est facile à gérer.

Le modèle en cascade est affecté par les inconvénients suivants :

Rigidité

L'inflexibilité linéaire du modèle rend difficile la mise en œuvre de tout changement après la fin de la phase d'élaboration des exigences.

Tests tardifs

Au cours de la phase de développement, les tests sont aléatoires ou absents, ce qui peut conduire à une identification tardive de problèmes importants.

Hypothèse d'exigences parfaites

Le modèle impose et suppose que chaque exigence est correcte et claire, ce qui peut s'avérer très irréaliste. Le modèle ne prévoit pas de vérification pendant le développement pour s'assurer que les exigences sont exactes et comprises par les développeurs.

Manque de retour d'information

Ce n'est qu'au cours de la dernière phase que le client peut s'exprimer sur le produit. Ainsi, si quelque chose (ou beaucoup de choses) ne répond pas à ses attentes, le problème n'apparaît qu'à la toute fin.

Développement logiciel agile, Scrum et Kanban

Pour explorer cet ensemble de modèles de développement de logiciels plus modernes, commençons par le mot *agile* : Que signifie-t-il ? L'agilité exprime la capacité à réagir et à se déplacer rapidement, tant dans le monde physique que mental. L'objectif est le même pour le développement de logiciels.

Le *développement logiciel agile* est une méthodologie qui s'appuie sur le développement itératif, la flexibilité et la collaboration. Elle se concentre sur l'apport de petites améliorations, tout en recueillant des commentaires et en les mettant en œuvre au cours du développement. Cette approche permet des réactions, des ajustements et des réponses rapides, ce qui permet de gagner du temps et de s'assurer que le projet répond aux attentes des utilisateurs et des clients.

Le mouvement agile a été lancé en 2001 par un *Manifeste pour le développement Agile de logiciels* (*Manifesto for Agile Software Development*) en ligne qui énonçait ses principes comme suit :

Nous découvrons comment mieux développer des logiciels par la pratique et en aidant les autres à le faire.

Ces expériences nous ont amenés à valoriser :

- *Les individus et leurs interactions* plus que les processus et les outils
- *Des logiciels opérationnels* plus qu'une documentation exhaustive
- *La collaboration avec les clients* plus que la négociation contractuelle
- *L'adaptation au changement* plus que le suivi d'un plan

Nous reconnaissons la valeur des seconds éléments, mais privilégions les premiers.

— Manifeste pour le développement Agile de logiciels, Traduction : agilemanifesto.org

Scrum

Scrum est l'un des cadres les plus populaires — peut-être le plus populaire — dans le domaine du développement agile de logiciels. Scrum repose sur les éléments suivants :

En bref, Scrum a besoin d'un *Scrum Master* pour favoriser un environnement où :

1. Un *Product Owner* ordonne le travail à faire pour résoudre un problème complexe dans le *Product Backlog*.
2. La *Scrum Team* transforme une sélection de ce travail en un Increment de valeur lors d'un *Sprint*.
3. La *Scrum Team* et ses parties prenantes inspectent les résultats et s'adaptent pour le prochain *Sprint*.
4. Répéter

— Le Scrum Guide (2020), Traduction : scrumguides.org

Mais qui sont le scrum master (chef de mêlée) et le product owner (propriétaire de produit) ? Qu'est-ce qu'un product backlog (carnet du produit) et un sprint (pointe de vitesse) ? Plongeons dans les rôles et les processus.

Sprints et planification des sprints

Un *sprint* est une phase de développement qui dure généralement de deux à quatre semaines et au cours de laquelle certaines tâches et, ou caractéristiques préalablement convenues sont réalisées. La *planification du sprint* intervient pour obtenir un accord sur les tâches. C'est au cours de ce processus que l'équipe prend des décisions sur les tâches qui peuvent être accomplies au cours du sprint à venir.

Ces choix sont basés sur les priorités fixées par le *propriétaire de produit* (*product owner* - PO), qui

contrôle le projet et en assure souvent le financement.

Carnet du produit et carnet de sprint

Comme nous venons de l'expliquer, le PO gère la liste des priorités, qui contient tout le travail souhaité sur le projet. Dans Scrum, cette liste est appelée le *carnet du produit* (*product backlog*). Chaque *carnet de sprint* (*sprint backlog*) contient les tâches sélectionnées pour le sprint en cours à partir du *product backlog*. Le *sprint backlog* contient les éléments qui ont été choisis par l'équipe lors de la planification du sprint.

Scrum quotidien ou Stand-up

Les *mêlées quotidiennes* (*daily Scrum*) ou *stand-up* sont de courtes réunions efficaces au cours desquelles les équipes discutent de leurs progrès, mettent en évidence les problèmes et les blocages, et partagent leurs plans pour la journée. Un scrum se tient généralement au tout début de la journée de travail.

Incrément

Un *incrément* est un objectif atteint au cours d'un sprint. Chaque sprint doit donner lieu à un ou plusieurs incréments. L'exactitude de la tâche ou des caractéristiques réalisées par l'incrément doit être vérifiée par des tests.

Revue de sprint

La *revue de sprint* (*sprint review*) est une réunion pour inspecter les résultats du sprint. Ces réunions sont généralement plus que des démonstrations—l'objectif est d'obtenir un retour d'information basé sur les résultats de l'équipe Scrum. L'objectif étant d'atteindre le but du produit, les parties prenantes donnent leur avis et font des suggestions sur les adaptations futures. Le résultat de ces réunions peut être des changements ou des ajouts au product backlog.

Rétrospective de sprint

L'objectif de la *rétrospective de sprint* (*sprint retrospective*) est d'améliorer la qualité et l'efficacité—et pas seulement du produit actuel. La rétrospective doit révéler les tensions cachées au sein de l'équipe, tout manque d'information qui ralentit les processus, les dépendances extérieures qui devraient être réduites pour alléger le fardeau de l'équipe, etc. L'équipe discute de ce qui s'est bien passé, des problèmes qu'elle a rencontrés et des solutions trouvées (ou non) à ces problèmes.

Le résultat est une liste de problèmes assortis de solutions possibles, attribuée à la personne responsable.

Scrum Master

Toutes ces réunions sont organisées par le *Scrum master*. Chaque équipe Scrum en a besoin d'un. Il est chargé de faciliter les processus Scrum et d'aider l'équipe à continuer à avancer vers l'objectif du produit lorsque des problèmes sont rencontrés au cours des sprints. Le Scrum master ne se contente pas de guider les processus, il agit également en tant qu'accompagnateur pour assurer une communication efficace au sein de l'équipe.

Product Owner

Le *product owner* est responsable de l'optimisation de la valeur du produit créé par l'équipe Scrum. Ce rôle peut varier selon les organisations et les équipes. Même lorsqu'il délègue des tâches, le product owner reste responsable des résultats.

Pour réussir, il faut que l'organisation respecte les décisions du product owner, qui se reflètent dans le product backlog et dans l'incrément de la revue de sprint. Le product owner est une personne unique, qui représente les besoins des différentes parties prenantes dans le product backlog. Toute personne souhaitant modifier le carnet de produit doit convaincre le product owner de le faire. Le product owner définit et hiérarchise la vision et le carnet du produit, et veille à la transparence, à la visibilité et à la compréhensibilité du product backlog.

Le Scrum master et le product owner collaborent à la réussite du projet. Leur partenariat permet à l'équipe de développement de fournir des fonctionnalités de grande valeur de manière efficace et efficiente.

Développeurs

Les développeurs mettent en œuvre les exigences en suivant le sprint backlog convenu. Ils peuvent travailler de manière indépendante, mais il existe plusieurs occasions dans lesquelles ils peuvent collaborer : par exemple, la programmation en binôme ou la révision du code d'un autre développeur.

Évaluation du modèle Scrum

Scrum est devenu la méthode préférée des équipes logicielles, mais elle présente aussi des avantages et des inconvénients.

Le modèle Scrum présente les avantages suivants :

Flexibilité

Les processus sont flexibles et itératifs, ce qui permet d'apporter des modifications en fonction des retours d'information.

Implication des clients

Un retour d'information régulier de la part des parties prenantes garantit l'alignement sur les besoins des clients.

Amélioration de la qualité

Des tests et des contrôles fréquents améliorent la qualité des produits.

Collaboration au sein de l'équipe

Le modèle met l'accent sur le travail d'équipe et la communication par le biais de réunions quotidiennes et régulières.

Le modèle Scrum est affecté par les inconvénients suivants :

Discipline requise

Scrum exige une adhésion stricte à ses pratiques, ce qui peut s'avérer difficile.

Dérive des objectifs

Si le product backlog n'est pas bien géré, les demandes des clients risquent d'être de plus en plus nombreuses et de retarder la sortie du produit.

Confusion des rôles

Les rôles standard tels que le Scrum master et le product owner peuvent être mal compris ou mal mis en œuvre.

Intensité des ressources

Les réunions quotidiennes et les contrôles fréquents peuvent prendre beaucoup de temps.

Kanban

Kanban est un autre cadre agile populaire qui met l'accent sur la visualisation du travail, la gestion des flux et l'amélioration des processus. L'une des grandes différences entre Scrum et Kanban est que ce dernier n'exige pas d'itérations de durée fixe. Il rend donc la livraison continue plus flexible et permet d'équilibrer les différentes priorités de travail.

Nous verrons dans les sous-sections suivantes ce dont les équipes ont besoin pour un projet Kanban.

Tableaux Kanban

Un tableau Kanban est un outil visuel qui permet de suivre le déroulement du travail par étapes. Les colonnes principales et les plus importantes sont "À faire" ("To Do"), "En cours" ("In Progress")

et “Terminé” (“Done”). D’autres colonnes peuvent être ajoutées, mais ces trois colonnes principales doivent figurer sur le tableau. Cette visualisation aide les équipes à détecter les goulots d’étranglement et à connaître l’état d’avancement des tâches en un clin d’œil.

Limite des travaux en cours

Une *limite des travaux en cours* (*work in progress (WIP) limit*) permet d’éviter le multitâche et d’améliorer la concentration. Avec cette limite, il y a un point où plus aucune tâche ne peut être ajoutée à la colonne “En cours”. De cette manière, les développeurs ne seront pas surchargés de tâches et pourront se concentrer sur celles sur lesquelles ils travaillent actuellement.

Membres de l’équipe

Les membres de l’équipe sont responsables de l’accomplissement des tâches et de l’établissement d’un flux fluide dans le système Kanban. Ils collaborent pour établir des priorités dans le travail et pour mettre en évidence et traiter tout type de problème qui apparaît.

Responsable Kanban

Le *responsable Kanban* (*Kanban manager*) supervise le processus Kanban, en veillant à ce que l’équipe respecte les principes et les pratiques Kanban. Il facilite les réunions, surveille les flux de travail et aide à résoudre les problèmes éventuels.

Évaluation du modèle Kanban

Le concept de base de ce modèle est simple. Examinons les avantages et les inconvénients.

Le modèle Kanban présente les avantages suivants :

Flux de travail visuel

Les tableaux Kanban offrent une visibilité claire de l’état d’avancement des travaux.

Flexibilité

En l’absence d’itérations fixes, le modèle prend en charge la livraison continue avec une grande capacité d’adaptation.

Limitation des tâches

La limite WIP permet d’éviter que les membres de l’équipe ne soient surchargés et donc d’améliorer leur concentration et leur efficacité.

Amélioration continue

Le modèle encourage l’évaluation régulière et l’amélioration des processus.

Le modèle Kanban est affecté par les inconvénients suivants :

Absence de délais

En l'absence de délais fixes, la gestion du temps est difficile.

Moins de structure

Le modèle peut manquer de la structure dont certaines équipes ont besoin pour rester organisées.

Discipline requise

Les limites WIP et la gestion du tableau nécessitent une attention particulière.

Risque de simplification excessive

Les descriptions des tâches peuvent simplifier à l'extrême des projets complexes si elles ne sont pas mises en œuvre de manière réfléchie.

DevOps

DevOps est une approche de développement de logiciels qui intègre le développement (Dev) et l'administration des infrastructures (*operations* - Ops) afin d'améliorer la collaboration, l'efficacité et la rapidité de livraison. L'objectif principal de DevOps est de raccourcir le cycle de développement des logiciels et d'assurer une livraison continue avec une qualité logicielle élevée. DevOps met l'accent sur l'automatisation, l'intégration et la livraison continues (*Continuous Integration and Continuous Delivery* - CI/CD), ainsi que sur une collaboration étroite entre des équipes traditionnellement isolées.

L'automatisation est cruciale dans le DevOps pour des tâches telles que l'intégration du code, les tests, le déploiement et la gestion de l'infrastructure. L'automatisation des tâches répétitives réduit les erreurs, accélère les processus et permet aux équipes de se concentrer sur des travaux plus stratégiques.

La surveillance et la journalisation continues sont essentielles pour maintenir la santé et les performances du système. Les pratiques DevOps comprennent la mise en place de systèmes complets de surveillance et de journalisation pour détecter les problèmes, analyser les tendances et améliorer la fiabilité du système.

Évaluation du modèle DevOps

Bien que le DevOps soit fortement recommandé pour de nombreux types de projets logiciels modernes, les avantages et les inconvénients doivent encore être pris en compte.

Le modèle DevOps présente les avantages suivants :

Rapidité et efficacité

Le modèle accélère les cycles de développement et de déploiement grâce à l'automatisation.

Collaboration améliorée

Le modèle comble le fossé entre les équipes de développement et d'administration.

Livraison continue

Ce modèle garantit que le logiciel est toujours en état d'être déployé, ce qui permet d'augmenter la fréquence des mises à jour.

Fiabilité élevée

L'automatisation des tests et de la surveillance améliore la fiabilité et réduit les erreurs.

Le modèle DevOps est affecté par les inconvénients suivants :

Transformation culturelle

Le modèle nécessite un changement culturel important et l'adhésion de l'ensemble de l'organisation.

Complexité

La gestion des flux CI/CD et de l'infrastructure automatisée peut s'avérer complexe.

Risques pour la sécurité

Le déploiement continu peut introduire des vulnérabilités en matière de sécurité s'il n'est pas géré avec soin.

Saturation d'outils

Le DevOps peut être accablant en raison de la multitude d'outils et de technologies impliqués.

Exercices guidés

1. Que signifie l'agilité ?

2. Quelle est la définition de Scrum, selon le Guide Scrum ?

3. Pourquoi Scrum est-il plus performant que le modèle en cascade en ce qui concerne le retour d'information des clients ?

4. Quels sont les aspects positifs du DevOps ?

Exercices d'exploration

1. Pourquoi le modèle en cascade n'est-il pas le meilleur à utiliser dans un environnement en évolution rapide ?

2. Que peut-il se passer lorsque le rôle du Scrum master est mal mis en œuvre ?

Résumé

La pertinence de la gestion de projets dans le développement de logiciels, en particulier dans les projets open source, réside dans sa capacité à fournir une structure, à améliorer la communication, à définir les rôles, à gérer les risques et à garantir la qualité. Ces éléments sont essentiels pour mener à bien les projets et favoriser un environnement de développement collaboratif et productif.

Dans cette leçon, nous avons passé en revue le modèle en cascade, les méthodologies agiles et DevOps. La connaissance de ces modèles est essentielle pour comprendre la gestion de projets dans le domaine du développement de logiciels. Il n'existe pas de méthode de travail idéale - chaque projet est différent. Dans cette leçon, vous avez appris les rôles, les processus et les avantages et inconvénients des différentes approches, ce qui peut vous aider à l'avenir à comprendre et peut-être à choisir le bon modèle pour un projet.

Réponses aux exercices guidés

1. Que signifie l'agilité ?

L'agilité exprime la capacité de réagir et de se déplacer rapidement, tant dans le monde physique que mental.

2. Quelle est la définition de Scrum, selon le Guide Scrum ?

- Un *Product Owner* ordonne le travail à faire pour résoudre un problème complexe dans le *Product Backlog*.
- La *Scrum Team* transforme une sélection de ce travail en un Increment de valeur lors d'un *Sprint*.
- La *Scrum Team* et ses parties prenantes inspectent les résultats et s'adaptent pour le prochain *Sprint*.
- Répéter

3. Pourquoi Scrum est-il plus performant que le modèle en cascade en ce qui concerne le retour d'information des clients ?

Le retour d'information étant recueilli au cours du développement, le produit final peut mieux répondre aux attentes du client.

4. Quels sont les aspects positifs du DevOps ?

Rapidité et efficacité — Collaboration améliorée — Livraison continue — Fiabilité élevée

Réponses aux exercices d'exploration

1. Pourquoi le modèle en cascade n'est-il pas le meilleur à utiliser dans un environnement en évolution rapide ?

Le modèle en cascade ne recueille des informations en retour qu'à la fin du développement. Ainsi, toute exigence mal comprise par les développeurs doit être corrigée à la toute fin. Si l'environnement change très rapidement, ce modèle ne doit pas être utilisé, car il ne prévoit pas de retour d'information au milieu du cycle de développement.

2. Que peut-il se passer lorsque le rôle du Scrum master est mal mis en œuvre ?

Un rôle de Scrum Master mal mis en œuvre peut entraver la capacité de l'équipe à fournir efficacement des produits de haute qualité et peut saper les avantages de l'adoption de Scrum. L'équipe peut manquer de conseils appropriés sur les pratiques et les principes de Scrum, ce qui conduit à une application incohérente ou incorrecte du cadre.

En l'absence d'un Scrum Master efficace pour éliminer les obstacles, ces derniers peuvent ralentir les progrès et réduire la productivité de l'équipe. Une mauvaise communication entre les membres de l'équipe et les parties prenantes peut se produire, entraînant des malentendus et des attentes mal alignées. L'équipe peut voir son moral et sa motivation baisser en raison de problèmes non résolus, d'un manque de soutien et d'une facilitation inefficace des événements Scrum. Des inefficacités peuvent résulter de réunions mal conduites, d'un manque de concentration et d'une planification et de revues de sprint inefficaces.



055.2 Gestion des produits / Gestion des versions

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 055.2

Valeur

2

Domaines de connaissance les plus importants

- Comprendre les types de versions les plus courants
- Comprendre la gestion des versions de logiciels et les raisons des versions majeures ou mineures
- Comprendre le cycle de vie d'un produit logiciel, depuis sa planification, son développement et sa publication jusqu'à sa mise hors service
- Comprendre la documentation relative aux versions d'un produit

Liste partielle de termes, fichiers et utilitaires utilisés

- Versions alpha et bêta
- Versions candidates
- Gel des fonctionnalités
- Versions majeures et mineures
- Gestion sémantique des versions
- Feuilles de route et étapes
- Journal des modifications (Changelog)
- Support à long terme (LTS)
- Fin de vie (EOL)

- Rétrocompatibilité



Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	055 Gestion de projets
Objectif :	055.2 Gestion des produits / Gestion des versions
Leçon :	1 sur 1

Introduction

La plupart des logiciels évoluent de différentes manières au fil du temps. En fait, c'est l'un des avantages des logiciels : ils sont faciles à modifier et il suffit d'un transfert réseau pour mettre à jour tous les utilisateurs du produit. Les développeurs ajoutent donc de nouvelles fonctionnalités, corrigent les bogues et les failles de sécurité, adaptent le logiciel à de nouveaux matériels et ajoutent des interfaces aux programmes et services populaires. À mesure que le logiciel évolue, de nouvelles *versions* ou *révisions* sont publiées.

Cette leçon traite de la logistique nécessaire à la planification et à la modification des logiciels, de la manière dont les équipes gèrent plusieurs versions, des conventions de dénomination des versions et d'autres aspects organisationnels du développement qui font partie des activités liées à la *gestion de projet*. La logistique qui s'applique spécifiquement au calendrier, à la dénomination et au contrôle des versions est connue sous le nom de *gestion des versions*.

Caractéristiques des versions

Les versions varient en termes de stabilité, de compatibilité ascendante et d'assistance. Nous

examinerons chacun de ces concepts dans les sections suivantes.

Versions stables et instables

La *stabilité* est une caractéristique essentielle que les utilisateurs attendent d'un logiciel. La première question que beaucoup se posent lorsqu'ils entendent parler d'une nouvelle version est "Est-elle stable ?". En d'autres termes, fonctionnera-t-elle de manière fiable ou risque-t-elle de planter, d'endommager les données ou de produire des résultats erronés ?

La stabilité peut être mesurée sur un spectre, et beaucoup de gens sont satisfaits d'utiliser un logiciel même s'il comporte encore quelques bogues. Mais les équipes de développement ont tendance à simplifier les choses et à parler en termes binaires de versions *stables* et *instables*.

La stabilité fait référence à la fois aux bogues du logiciel et à sa propension à changer de manière visible pour les utilisateurs externes. Les développeurs peuvent considérer qu'une version d'une bibliothèque logicielle est instable parce qu'ils ont modifié les fonctions appelées par les programmeurs (c'est-à-dire parce que les programmeurs pourraient devoir réécrire leur logiciel pour la prochaine version). Ou bien, un logiciel peut être instable du point de vue de l'utilisateur parce que les développeurs ont supprimé une fonctionnalité.

Y a-t-il une raison de publier une version instable ? Oui : elles sont utiles pour permettre aux utilisateurs d'essayer les nouvelles fonctionnalités dès le début du projet et de les tester afin de détecter les bogues.

La plupart des projets publient très tôt des versions préliminaires du logiciel à l'intention de leurs principaux clients afin qu'ils puissent les tester ; ces versions sont appelées versions *alpha*. Personne ne devrait utiliser ces versions pour un travail réel — elles sont uniquement destinées à des fins de test. En fait, les testeurs devraient exécuter les versions alpha sur des ordinateurs qui ne sont pas utilisés pour des tâches importantes, car les bogues présents dans ces versions pourraient endommager les données stockées sur l'ordinateur.

Lorsque le logiciel est sur le point d'être considéré comme stable, le projet publie généralement une autre version de test appelée version *bêta*. Celle-ci ne doit toujours pas être utilisée pour le travail de production, mais uniquement à des fins de test.

Pour les versions alpha et bêta, les développeurs disposent d'un processus permettant de signaler les bogues et de suivre l'avancement de leur correction.

Certains projets comprennent une étape supplémentaire entre les versions bêta et stable, lorsque les développeurs ont corrigé tous les bogues possibles et estiment que le produit est terminé. Ils appellent cette version *version candidate* et la présentent à leurs principaux clients ou parties prenantes afin que ces derniers puissent planifier la manière dont ils utiliseront les nouvelles

fonctionnalités.

Enfin, certains projets intègrent des fonctionnalités qui ne sont pas encore tout à fait stables, car des utilisateurs importants les ont demandées. Les développeurs souhaitent que les utilisateurs testent ces fonctionnalités et donnent leur avis avant de finaliser les interfaces et d'investir des efforts pour les stabiliser.

Rétrocompatibilité

La plupart des projets visent la *rétrocompatibilité*, ce qui signifie qu'ils s'efforcent de ne pas supprimer les fonctionnalités ou fonctions qui existaient dans les versions antérieures. La compatibilité peut exister à plusieurs niveaux. Par exemple, si les développeurs maintiennent la rétrocompatibilité pour l'interface de programmation d'application (*Application Programming Interface* - API), ils garantissent que l'ancien code source continuera de fonctionner, mais qu'il devra éventuellement être recompilé. Si les fournisseurs de matériel ou les développeurs de systèmes d'exploitation maintiennent la rétrocompatibilité pour l'interface binaire d'application (*Application Binary Interface* - ABI), ils garantissent en outre que les anciens fichiers exécutables pourront être exécutés sur les nouvelles versions du matériel.

Nous examinerons plus tard comment les projets gèrent le *manque* de rétrocompatibilité lorsqu'ils décident que l'ancienne interface ne convient vraiment pas aux nouvelles fonctionnalités ou aux nouveaux environnements qu'ils doivent prendre en charge.

Support et fin de vie

Idéalement, les logiciels devraient s'améliorer au fur et à mesure que les développeurs découvrent et corrigent leurs problèmes. Mais avec le temps, certaines fonctionnalités peuvent cesser de fonctionner en raison de changements dans l'environnement du logiciel. De plus, de nouveaux bogues, auparavant cachés, sont découverts.

Les nouvelles versions combinent souvent des corrections de sécurité et de bogues avec de nouvelles fonctionnalités. Vous ne souhaitez peut-être pas bénéficier des nouvelles fonctionnalités (en fait, vous préférez peut-être certaines anciennes fonctionnalités que les développeurs ont supprimées pour faire place à de nouvelles), mais les utilisateurs passent généralement à la nouvelle version pour bénéficier des correctifs de sécurité qui les protègent contre les attaques malveillantes.

Certaines personnes continuent d'utiliser d'anciennes versions de logiciels, refusant de passer à une version plus récente. Cela s'explique généralement par le fait que la nouvelle version perturbe le fonctionnement de leur environnement. Lorsque les entreprises facturent les mises à jour, certains clients refusent de les installer, car ils ne veulent pas payer de supplément, mais

payer pour obtenir des mises à jour est très rare dans le domaine de l'open source.

Les développeurs accommodent les utilisateurs des anciennes versions, dans la mesure du possible, en corrigeant les bogues dans ces versions, sans ajouter de fonctionnalités qui pourraient endommager le logiciel. Naturellement, les développeurs ne peuvent pas faire cela indéfiniment, car cela leur prend du temps et de l'énergie qu'ils pourraient consacrer à de nouveaux travaux. Il arrivera un moment où ils refuseront de corriger une ancienne version et diront aux utilisateurs de mettre à niveau leur logiciel ou d'apporter eux-mêmes les corrections nécessaires.

La correction des bogues et des failles de sécurité est appelée *support* pour le logiciel. Cela diffère du support offert par les services d'assistance et les autres personnes qui aident les utilisateurs à comprendre le logiciel. En ce qui concerne la gestion des versions, une *version supportée* est une version pour laquelle les développeurs s'engagent à corriger les bogues, tandis qu'une *version non supportée* est une version pour laquelle ils ne s'engagent pas à le faire.

Notez que la notion de "support" s'applique principalement aux logiciels créés par des entreprises ou de grandes organisations. Les projets open source de plus petite envergure, qui dépendent fortement de bénévoles, promettent souvent de faire de leur mieux pour corriger les bogues et publier de nouvelles versions. Ils ne voient pas la nécessité de corriger les anciennes versions. Comme le code est open source, les utilisateurs qui ne souhaitent pas effectuer de mise à niveau peuvent payer quelqu'un pour corriger une ancienne version.

Lorsque le support est en place, les développeurs publient un calendrier indiquant la durée pendant laquelle ils assureront le support de chaque version. La date à laquelle le support prend fin est appelée *fin de vie* (*End Of Life* - EOL) du logiciel. Les utilisateurs peuvent conserver l'ancienne version jusqu'à la fin de vie et s'attendre à ce que les bogues soient corrigés, mais après la fin de vie, ils doivent effectuer une mise à niveau ou prendre le risque de continuer à utiliser la version antérieure.

Les projets de grande envergure, tels que la distribution Debian de GNU/Linux, proposent des versions avec un *support à long terme* (*Long Term Support* - LTS). Cela signifie simplement que les développeurs continueront à corriger les bogues pendant un certain nombre d'années. Les clients qui accordent une grande importance à la stabilité peuvent hésiter à installer des versions de logiciels comportant de nombreuses modifications, de peur que celles-ci ne perturbent les processus dont ils dépendent. Ces clients, notamment les grandes institutions, apprécient la sécurité offerte par les versions LTS.

Versions logicielles : majeures, mineures et correctifs

Nous avons vu que la gestion des versions est complexe : certaines versions corrigent des bogues,

tandis que d'autres ajoutent des fonctionnalités, et leur stabilité peut varier. Les développeurs tentent d'indiquer à l'aide d'étiquettes l'importance des modifications apportées au logiciel dans chaque version. Les conventions adoptées par la quasi-totalité des projets pour étiqueter les versions sont appelées *gestion sémantique des versions*.

Prenons l'exemple des débuts du noyau Linux. Linus Torvalds a baptisé la première version stable 1.0. Au fur et à mesure qu'il améliorait le noyau, il a baptisé les versions suivantes 1.1, 1.2, etc. Le 1 initial correspondait à la *version majeure* et les chiffres après le point indiquaient la *version mineure*. Vous pouvez supposer que la version 1.2 offrait plus de fonctionnalités et plus d'avantages que la version 1.1.

Mais il y a eu également d'innombrables petites versions, parfois juste pour corriger quelques bogues. Afin de montrer que les modifications avaient un impact mineur sur l'utilisation du noyau, Torvalds a ajouté un troisième chiffre appelé *correctif (patch)*. Ainsi, la version 1.0 a été mise à jour en 1.0.1, puis en 1.0.2, et ainsi de suite.

Les numéros de correctif recommencent à 0 lorsqu'une nouvelle version mineure est publiée, et les numéros de version mineure commencent à 0 lorsqu'une nouvelle version majeure est publiée.

Les développeurs regroupent les versions à l'aide d'un "x" pour indiquer qu'ils font référence à plusieurs versions, par exemple 1.x pour toutes les versions de la version majeure 1.

Mais quelle est la différence entre un changement qui conduit à une nouvelle version mineure et un changement qui mérite de déclencher une nouvelle version majeure ? En général, plusieurs années s'écoulent avant la sortie d'une nouvelle version majeure, qui doit refléter une mise à niveau très importante.

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 one, 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.

Cycle de vie d'un logiciel

Ne croyez pas que la vie d'un développeur soit facile. Tout le monde veut quelque chose de lui. Je veux que ce bogue dans la composition graphique sur l'écran soit corrigé dès que possible, tandis

que quelqu'un d'autre dit que le bogue dans le nommage des fichiers est plus important. Les utilisateurs réclament de nouvelles fonctionnalités, et lorsque différents développeurs travaillent en parallèle sur différentes fonctionnalités, ils se rendent compte que les modifications apportées par l'un peuvent aller à l'encontre de celles apportées par un autre.

La gestion de projet, et le sous-ensemble de ces tâches appelé gestion des versions, traitent ces questions. En général, un membre senior du projet assume la responsabilité de cette tâche de gestion.

Tout comme les personnes, les versions logicielles ont un cycle de vie. Chacune commence par une discussion sur les nouvelles fonctionnalités et autres changements nécessaires, passe par des phases de développement et de test, puis est déployée de manière planifiée.

Planification et feuilles de route

Les développeurs essaient de définir, bien à l'avance, les modifications qu'ils souhaitent apporter à un produit. Dans les entreprises commerciales, ils discutent avec les responsables marketing, qui filtrent et résument les commentaires des clients. Les projets open source dépendent davantage des idées soumises par les développeurs et les utilisateurs. Ces idées sont enregistrées dans une base de données appelée *suivi des problèmes (issue tracker)*. Si vous avez besoin qu'un bogue soit corrigé ou si vous souhaitez une nouvelle fonctionnalité, vous rédigez un *problème*. Les développeurs hiérarchisent ensuite les modifications et décident qui se chargera de chacune d'entre elles.

Alors, comment les changements sont-ils choisis et classés par ordre de priorité ? Le processus peut s'avérer compliqué. Mais les bons chefs de projet open source encouragent la participation de tous tout en veillant à ce que les décisions soient prises. Certains projets organisent même des conférences où les participants discutent des priorités.

Une *feuille de route* publiée présente les plans d'amélioration et de modification qui en résultent. Elle peut s'étendre sur plusieurs versions et plusieurs années. Chaque phase est appelée *étape (milestone)* et peut être associée ou non à une date cible.

Planification des versions

Il existe deux méthodes classiques pour planifier les versions : la méthode basée sur le temps et celle basée sur les fonctionnalités. Un projet peut promettre une version à intervalles réguliers, par exemple tous les six mois—et inclure tout ce qui est terminé à ce moment-là. Alternativement, le projet peut promettre d'inclure certaines fonctionnalités dans une version et laisser les développeurs prendre le temps nécessaire pour terminer ces fonctionnalités.

À l'approche de la date de sortie, le responsable de la publication détermine les dates des versions

alpha, bêta et stable. Les membres de l'équipe examinent régulièrement les rapports de bogues et s'efforcent de planifier leur travail de manière à respecter ces échéances.

Pour finaliser une version, un projet doit cesser d'accepter des nouvelles fonctionnalités et se concentrer sur le bon fonctionnement des fonctionnalités existantes. Ce moment est appelé *gel des fonctionnalités (feature freeze)*.

Documentation pour les versions du produit

Comme mentionné précédemment, les feuilles de route expliquent les fonctionnalités que les développeurs prévoient d'inclure dans chaque version. La version est accompagnée d'une liste de modifications appelée *journal des modifications (changelog)*. En général, le journal des modifications répertorie les nouvelles fonctionnalités, les modifications apportées aux fonctionnalités existantes, les fonctionnalités qui ont été supprimées, les fonctionnalités que les développeurs ont l'intention de supprimer à l'avenir (appelées fonctionnalités *obsolètes*) et les corrections de bogues.

Ainsi, les journaux des modifications peuvent être assez longs et détaillés. Les utilisateurs doivent porter une attention particulière aux fonctionnalités qui ont été supprimées ou qui sont obsolètes, car ils pourraient devoir modifier leurs programmes ou leur façon d'utiliser le produit. Évidemment, les développeurs s'efforcent de ne supprimer que les fonctionnalités dont plus personne n'a besoin.

Chaque nouvelle version doit entraîner une modification de la documentation du produit afin de refléter les changements apportés à celui-ci. Cette tâche peut prendre beaucoup de temps, et il est facile pour les développeurs de passer à côté d'une modification ou de prendre du retard dans la rédaction de la documentation.

Exercices guidés

1. Qu'est-ce qui distingue une version stable d'une version instable ?

2. Vous attendez-vous à un changement de fonctionnalités entre une version nommée 2.6.14 et une version nommée 2.6.15 ?

3. Vous attendez-vous à ce qu'il y ait des changements de fonctionnalités entre une version nommée 2.6.0beta et une version nommée 2.6.0 ?

4. Pourquoi vous attendez-vous à ce que la version 1.0 ne soit pas rétrocompatible avec la version 0.9 ?

5. Si vous découvrez une faille de sécurité dans une version après le gel des fonctionnalités mais avant sa sortie, pouvez-vous faire corriger cette faille ?

Exercices d'exploration

1. Supposons que vous souhaitiez continuer à utiliser une version d'un logiciel open source après sa fin de vie, car elle dispose d'une fonctionnalité dont vous avez besoin. Que pouvez-vous faire pour continuer à l'utiliser ?

2. Quels sont les critères qui permettent de choisir une correction de bogue ou une demande de fonctionnalité plutôt qu'une autre ?

Résumé

Cette leçon a décrit les principales caractéristiques qui distinguent les versions : stabilité, rétrocompatibilité et support. Nous avons abordé la signification des noms et des numéros de version, ainsi que les aspects clés de la gestion des versions, y compris la documentation.

Réponses aux exercices guidés

1. Qu'est-ce qui distingue une version stable d'une version instable ?

Les versions sont considérées comme stables à deux égards : elles fonctionnent correctement sans planter ni produire de résultats erronés, et l'interface présentée aux utilisateurs ou aux programmeurs est censée être rétrocompatible avec les versions précédentes.

2. Vous attendez-vous à un changement de fonctionnalités entre une version nommée 2.6.14 et une version nommée 2.6.15 ?

Non. Dans la gestion sémantique des versions, le troisième chiffre indique un correctif, qui a été apporté pour corriger un bogue ou effectuer une autre tâche mineure telle que le reformatage. Une modification de fonctionnalités doit donner lieu à une version mineure ou majeure.

3. Vous attendez-vous à ce qu'il y ait des changements de fonctionnalités entre une version nommée 2.6.0beta et une version nommée 2.6.0 ?

Non. La version bêta est une version de test qui contient toutes les fonctionnalités qui seront incluses dans la version finale.

4. Pourquoi vous attendez-vous à ce que la version 1.0 ne soit pas rétrocompatible avec la version 0.9 ?

Le numéro 0.9 avertit explicitement les utilisateurs potentiels que le logiciel est encore en cours de conception et qu'il est très probable que son interface soit différente lorsqu'il sera stabilisé en version 1.0.

5. Si vous découvrez une faille de sécurité dans une version après le gel des fonctionnalités mais avant sa sortie, pouvez-vous faire corriger cette faille ?

Tout à fait. La période entre le gel des fonctionnalités et la sortie du produit est destinée à permettre la détection et la correction des problèmes, y compris les failles de sécurité.

Réponses aux exercices d'exploration

1. Supposons que vous souhaitiez continuer à utiliser une version d'un logiciel open source après sa fin de vie, car elle dispose d'une fonctionnalité dont vous avez besoin. Que pouvez-vous faire pour continuer à l'utiliser ?

Après la fin de vie, les développeurs du projet ne sont plus tenus de corriger les bogues, y compris les failles de sécurité. Vous devez donc suivre assidûment le suivi des bogues et les listes de diffusion du projet pour connaître les bogues qui apparaissent. Comme le code est disponible, vous pouvez et devez corriger les bogues présents dans votre version. En théorie, vous pourriez même intégrer de nouvelles fonctionnalités dans votre version. Cela ferait alors de votre version une bifurcation (*fork*) de la version originale.

2. Quels sont les critères qui permettent de choisir une correction de bogue ou une demande de fonctionnalité plutôt qu'une autre ?

Pour les corrections de bogues, les critères comprennent la gravité (qui est attribuée au bogue après son enregistrement dans le système de suivi des bogues) et le nombre d'utilisateurs concernés. Pour une fonctionnalité, les critères comprennent le nombre d'utilisateurs qui souhaitent cette fonctionnalité, la difficulté de son codage et son impact potentiel sur d'autres parties du programme.



055.3 Gestion des communautés

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 055.3

Valeur

2

Domaines de connaissance les plus importants

- Comprendre les rôles dans les projets open source
- Comprendre les tâches courantes dans les projets open source
- Comprendre les différents types de contributions aux projets open source
- Comprendre les différents types de contributeurs aux projets open source
- Comprendre le rôle des organisations dans la maintenance des projets open source
- Comprendre le transfert des droits des individus vers une organisation chargée de la maintenance d'un projet
- Comprendre les règles et les politiques dans les projets de logiciels open source
- Comprendre l'attribution et la transparence des contributions
- Comprendre les aspects de la diversité, de l'équité, de l'inclusion et de la non-discrimination

Liste partielle de termes, fichiers et utilitaires utilisés

- Développement de logiciels
- Documentation
- Conceptions et illustrations
- Assistance aux utilisateurs
- Développeurs

- Responsables de publication
- Utilisateurs
- Chefs de projet et dictateurs bienveillants
- Particuliers et entreprises
- Passionnés et professionnels
- Membres de l'équipe principale et contributeurs occasionnels
- Contributions au code et à la documentation
- Rapports de bogues
- Bifurcations (Forks)
- Fondations et sponsors
- Accords de contribution
- Certificats d'origine des développeurs (DCO)
- Directives de codage
- Codes de conduite



Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	055 Gestion de projets
Objectif :	055.3 Gestion des communautés
Leçon :	1 sur 1

Introduction

L'une des principales raisons de créer un projet open source est d'attirer les contributions de nombreuses personnes différentes. L'open source permet de répondre plus facilement aux différents besoins et intérêts des utilisateurs du projet et de tirer parti de leurs compétences variées. Par conséquent, une communauté diversifiée est essentielle à la réussite du projet. C'est au sein de la communauté que les forces créatives se réunissent pour faire aboutir votre projet.

Cette leçon explique les éléments fondamentaux des communautés de logiciels libres et open source, ainsi que la manière dont leurs membres collaborent. Bien sûr, chaque communauté est unique, car elle est composée de personnes différentes et les projets ont des objectifs et des exigences variés.

Rôles dans les projets open source

La plupart des projets open source ont pour principal résultat des logiciels. Ils nécessitent donc au minimum des programmeurs, des concepteurs ou architectes logiciels, des testeurs, des responsables de publication et d'autres experts en développement de code. La documentation fait généralement partie intégrante de ces projets, ce qui signifie que la communauté a également

besoin d’auteurs, d’éditeurs et de réviseurs.

D’autres projets peuvent ne pas produire de code, mais d’autres “livrables” tels qu’un livre, un projet artistique ou musical, ou un rapport stratégique. Wikipédia est un exemple bien connu de collaboration sur un projet open source axé sur les documents textuels et les médias. Ces projets nécessitent des experts en conception, production et livraison des livrables.

Les communautés bénéficient également de nombreuses autres compétences générales, telles que le marketing et la promotion, l’administration, le conseil juridique, le talent artistique pour créer des logos ou des diagrammes dans la documentation, et les compétences nécessaires à la maintenance du site web du projet et d’autres outils de communication. Les projets open source peuvent organiser des rencontres physiques et même des conférences, auquel cas ils ont besoin d’organiseurs pour donner vie à ces événements.

Les paragraphes précédents vous donnent une idée des types de compétences recherchées par une communauté. Ces rôles généraux peuvent être subdivisés en diverses tâches spécifiques. Un exemple simple d’une telle tâche consiste pour certains rédacteurs à corriger le travail d’autres rédacteurs.

Parmi les programmeurs, il existe toute une gamme d’expériences. Un projet sain a toujours besoin de programmeurs *vétérans* (ou *seniors*) qui connaissent bien son code et ses normes de codage, ainsi que de *débutants* qui apportent de nouvelles idées et une aide simple, comme la correction de bogues. Certains débutants deviendront, espérons-le, la prochaine génération de vétérans.

La qualité du code nécessite un contrôle minutieux de ce qui est intégré au dépôt central partagé avec les utilisateurs. C’est pourquoi certains contributeurs chevronnés obtiennent le statut de *commiteur* (*committer*). Ils sont chargés de vérifier la qualité, l’utilité et la conformité aux normes de codage des contributions. Ils ont le privilège d’accepter le code proposé dans le dépôt fiable. Les autres contributeurs soumettent leur code aux commiteurs pour vérification.

De nombreux commiteurs et autres vétérans jouent également le rôle de *mentors* auprès des nouveaux contributeurs afin de leur enseigner les normes et les pratiques nécessaires pour que leur code soit accepté.

Certains commiteurs ne savent pas apprécier le cadeau qu’un contributeur leur offre. Si la contribution ne répond pas aux normes de qualité ou de codage, le commiteur peut tout simplement la rejeter. Mais ce rejet décourage le contributeur potentiel et élimine une précieuse opportunité d’apprentissage. N’oubliez pas que les bons commiteurs sont également des mentors. Ils donnent donc au contributeur des conseils sur la manière de mettre le code aux normes et travaillent avec lui au fil du temps. Si la contribution est vraiment inutilisable, le contributeur devrait au moins être remercié et encouragé à travailler sur autre chose pour le projet. Il est

important d'aider les gens à découvrir et à affiner leurs compétences.

Lorsqu'une entreprise lance un projet open source, elle désigne parfois des commiteurs parmi son propre personnel afin de s'assurer qu'elle peut contrôler l'orientation et la qualité du projet. Mais souvent, les entreprises autorisent des personnes extérieures à l'entreprise qui font preuve de compétences et de dévouement à devenir des commiteurs.

Les *chefs de projet* prennent des décisions cruciales, telles que les nouvelles fonctionnalités à prendre en charge. Ces chefs de projet doivent souvent prendre des décisions qui ne concernent pas le codage, telles que la manière de promouvoir le projet, de résoudre les désaccords majeurs et d'obtenir des financements. Les chefs de projet et les commiteurs travaillent en étroite collaboration avec les *gestionnaires de publication* (*release managers*) afin de déterminer quand le code source est stable et prêt à être partagé avec les utilisateurs.

Certains projets ont un seul chef de projet, souvent appelé *dictateur bienveillant*. Il s'agit généralement d'une personne qui a lancé le projet (Linus Torvalds en est un exemple bien connu dans le cas du noyau Linux) et qui jouit d'une grande autorité, voire d'un certain charisme. Cependant, la plupart des projets préfèrent mettre en place un petit comité de chefs de projet plutôt qu'un seul dictateur bienveillant. Même le projet Linux est passé à une approche par comité.

Les projets peuvent également demander à certains contributeurs d'apporter leur aide aux utilisateurs sur les forums du projet. Mais les projets sains devraient compter de nombreux utilisateurs compétents qui s'entraident.

Nous terminerons cette section par un rôle très important : celui du *gestionnaire de communauté* (*community manager*). Il s'agit d'une personne chargée de maintenir une communication ouverte et constructive et de veiller à ce que la communauté progresse vers son objectif. Le responsable de communauté sait comment accueillir et motiver les contributeurs, résoudre les conflits, protéger les membres de la communauté contre les abus et accomplir d'autres tâches visant à préserver la bonne santé de la communauté.

Tâches courantes dans les projets open source

À bien des égards, les projets open source impliquent les mêmes tâches que les projets traditionnels menés au sein d'une seule entreprise. Par exemple, tout code doit être testé. Mais à certains égards, les projets open source diffèrent des projets propriétaires, où seul un nombre limité de personnes sont autorisées à modifier le code et où le code source est souvent caché du public.

Par exemple, les entreprises affectent généralement du personnel qualifié chargé de l'assurance

qualité (AQ) pour tester le code. Mais de nombreux projets open source demandent simplement à leurs utilisateurs de tester chaque version. (Les projets propriétaires impliquent également leurs utilisateurs dans les tests, en plus de l'AQ.)

Autre exemple de différence : un projet propriétaire affecte généralement des développeurs rémunérés à des tâches spécifiques et leur indique le temps qu'ils doivent consacrer chaque semaine à ces tâches. Certains projets open source bénéficient de la contribution de personnes rémunérées par leur employeur, voire parfois employées par le projet lui-même, et qui se voient attribuer des tâches de la même manière que les développeurs propriétaires. Mais dans la plupart des projets sains, une grande partie des contributions provient de bénévoles qui effectuent les tâches dès qu'ils en ont le temps. Comme le projet ne dépend pas de tous les bénévoles pour être mené à bien dans les délais, une version open source peut être retardée jusqu'à ce que les développeurs estiment que les fonctionnalités sont prêtes, ou peut être publiée à des dates fixes avec les fonctionnalités qui sont prêtes.

La communication est essentielle tant pour les projets propriétaires que pour les projets open source, mais elle s'effectue souvent de manière différente. Les projets propriétaires rassemblent souvent une équipe dans un même bureau et organisent régulièrement des réunions (*stand-up meetings*) selon une méthodologie de développement telle que SCRUM. Les projets open source étant géographiquement dispersés, ces méthodologies sont rares. La communication s'effectue plutôt en ligne et de manière asynchrone.

En résumé, les projets de logiciels libres et open source atteignent souvent les mêmes objectifs que les projets propriétaires, mais de manière différente.

Le signalement des bogues est un élément essentiel du développement logiciel qui donne lieu à la création d'un ensemble de rôles spécifiques. Quelqu'un doit surveiller la base de données des bogues et attribuer des priorités. Si un bogue est critique (et en particulier s'il peut affaiblir la sécurité du logiciel), il doit être attribué à un développeur compétent afin qu'il le corrige.

Les chefs de projet et les gestionnaires de communauté doivent examiner la participation au projet et identifier les lacunes. Y a-t-il trop peu de personnes disposées à tester le code ? La documentation est-elle incomplète ? Trop de vétérans ou de membres seniors du projet quittent-ils sans être remplacés ? Les chefs de projet et les gestionnaires de communauté peuvent se concentrer sur le recrutement de personnes pour pourvoir les postes nécessaires.

Les chefs de projet et les gestionnaires de communauté travaillant sur des projets de grande envergure collectent souvent eux aussi des indicateurs afin d'obtenir des informations importantes, par exemple pour savoir si les bogues importants sont corrigés en temps opportun.

Types de contributions dans l'open source

Comme expliqué dans une section précédente, les personnes qui ajoutent du code, de la documentation ou d'autres éléments dans le dépôt central sont appelées *contributeurs*. Les membres du projet qui approuvent les contributions et les acceptent dans le dépôt sont appelés *commiteurs*. Certains membres assument d'autres rôles courants dans le développement de logiciels.

Bien que le terme *contributeur* soit généralement réservé à une personne qui développe du code ou une autre partie de l'offre officielle du projet, toute personne qui participe à la réussite du projet apporte une contribution d'une manière ou d'une autre et mérite d'être remerciée pour cela. Ces contributions plus informelles comprennent le signalement d'un bogue, la réponse à une question sur un forum, un don ou une collecte de fonds, et la promotion du projet auprès de personnes extérieures.

Les projets open source, tout comme les projets propriétaires, demandent aux utilisateurs ce qu'ils souhaitent en termes de fonctionnalités, d'améliorations des performances ou d'autres modifications du projet. Ces utilisateurs apportent une contribution importante en exprimant leurs opinions.

Types de contributeurs dans l'open source

De nombreuses communautés obtiennent des contributions non seulement de particuliers, mais aussi d'entreprises qui rémunèrent leurs employés pour qu'ils contribuent à un projet que l'entreprise juge important pour ses activités.

Parfois, le fait d'avoir des contributeurs rémunérés aux côtés de bénévoles peut créer des tensions. Les bénévoles peuvent avoir l'impression que leur travail est exploité ou craindre que les contributeurs rémunérés tentent d'orienter le projet dans le sens des intérêts de leurs employeurs. Le gestionnaire de communauté et les chefs de projet doivent garantir que toutes les contributions acceptées profitent à l'ensemble du projet et à ses utilisateurs. Les bénévoles doivent également être motivés à contribuer pour des raisons personnelles, que ce soit par loyauté envers la communauté ou pour répondre à leurs propres besoins.

Certaines personnes contribuent régulièrement à un projet et assument des responsabilités à long terme ; on les appelle les *membres de l'équipe principale* (*core team members*). Les communautés saines comptent également des contributeurs occasionnels qui signalent des bogues ou publient des conseils sur les forums, mais qui ne sont pas censés assumer la responsabilité du projet.

De même, certains contributeurs seront des professionnels dans leur domaine — qu'ils soient rémunérés ou non par une entreprise pour travailler sur le projet — tandis que d'autres seront

des amateurs, souvent appelés *enthousiastes* (*enthusiasts*). Mais il n'est pas nécessaire d'être un professionnel pour jouer un rôle important. Vous pourriez même devenir un membre de l'équipe principale ou chef de projet.

Rôle des organisations dans les projets open source

Bien que de nombreux projets open source soient créés par des individus passionnés ou de petits groupes de bénévoles, ceux-ci recherchent généralement le soutien d'organisations lorsque les projets prennent de l'ampleur. Ce soutien peut prendre différentes formes. En général, les organisations apportent leur contribution parce qu'un projet open source leur permet de répondre à leurs besoins commerciaux de manière plus économique et plus fiable que de réinventer les mêmes fonctionnalités dans un code propriétaire. Les garanties offertes par une licence libre ou open source sont appréciées par de nombreuses organisations.

Certains idéalistes se méfient des entreprises ou autres grandes organisations et préféreraient que les projets open source restent totalement exempts de leur influence. Au fil des ans, cette attitude plutôt simpliste est devenue moins courante. La plupart des défenseurs de l'open source estiment que les entreprises et autres organisations peuvent apporter un soutien essentiel aux projets open source.

De nombreux projets open source voient le jour au sein d'une organisation et peuvent même être basés sur du code propriétaire que l'entreprise décide d'ouvrir. Pour gagner de l'argent, l'entreprise peut décider de garder un contrôle strict sur le projet et de le diviser en deux parties, l'une ouverte et l'autre propriétaire : c'est ce qu'on appelle le modèle *open core*. De nombreux fondateurs de projets commencent par l'open source, mais créent ensuite une entreprise autour de celui-ci, qui peut ou non utiliser le modèle open core.

D'autres projets s'efforcent de garantir qu'aucune entreprise ne contrôle à elle seule leur orientation. Pour préserver leur indépendance, ils doivent généralement s'assurer le soutien de plusieurs organisations, afin qu'aucune d'entre elles ne soit suffisamment puissante pour prendre des décisions de manière dictatoriale.

Comment les organisations soutiennent-elles l'open source ? Comme mentionné précédemment, beaucoup affectent du personnel rémunéré à des projets. En outre, elles peuvent embaucher des personnes très productives qui ont contribué au projet en tant que bénévoles et ont développé une expertise dans ce domaine. Cette voie vers l'emploi est un moyen précieux pour les étudiants et autres contributeurs bénévoles de faire progresser leur carrière.

Les entreprises qui souhaitent des extensions qui n'intéressent pas les autres utilisateurs ne devraient pas faire pression sur le projet open source pour qu'il ajoute ces fonctionnalités supplémentaires, mais devraient payer leur personnel pour les développer sans les reverser au

projet.

Un exemple intéressant de la tension potentielle créée par les besoins des entreprises est illustré par le célèbre système d'exploitation Android, basé sur Linux et développé par Google pour équiper ses appareils mobiles. Certaines modifications apportées à Linux par Google sont reversées à la communauté Linux, tandis que d'autres modifications n'apparaissent que dans Android. Les développeurs Linux rejettent parfois les modifications qui leur sont soumises par Google, tout comme tous les projets choisissent les contributions à intégrer.

Il existe de nombreuses raisons pour lesquelles une entreprise ou un groupe de développeurs peut créer une version distincte de votre code. Idéalement, ils peuvent répondre à leurs besoins en ajoutant une bibliothèque optionnelle ou une séquence de code qui peut être exclue lors de la compilation. Mais parfois, un groupe ressent un besoin important qui est incompatible avec l'orientation choisie par les responsables du projet. Lorsqu'une version distincte est créée, on parle alors de *bifurcation* (*fork*).

Les bifurcations étaient autrefois considérées comme des échecs de collaboration, mais aujourd'hui, elles sont beaucoup mieux acceptées. En général, les personnes qui créent une bifurcation mettent en place un nouveau dépôt et lancent un nouveau projet. Certaines personnes travaillent à la fois sur le projet original et sur la bifurcation.

(Notez que GitHub utilise le mot “fork” pour désigner un phénomène très différent : créer un clone ou une copie du code d'un projet afin de travailler dessus séparément.)

En plus du code, de nombreuses entreprises apportent une contribution financière. Elles peuvent financer des initiatives telles que le marketing et les conférences. Elles peuvent rejoindre le conseil d'administration et offrir des conseils d'experts sur l'orientation et la stratégie du projet.

Un exemple illustrant l'importance d'un tel “soutien souple” provient du serveur web historique Apache. Le projet a fait un grand bond en avant dès ses débuts lorsque IBM a manifesté son intérêt. Les avocats d'IBM ont montré au projet comment mettre en place des garanties juridiques, et une réunion financée par IBM a aidé les dirigeants d'Apache à créer une organisation solide.

Pour préserver leur indépendance, de nombreux projets open source créent une fondation à but non lucratif ou rejoignent une fondation existante. Parmi les exemples célèbres de fondations qui encadrent des projets open source, on peut citer la Linux Foundation, l'Apache Foundation et l'Eclipse Foundation.

Le soutien d'une fondation est précieux car il permet de fournir les services logistiques dont la plupart des développeurs de logiciels ne veulent pas s'occuper : assistance juridique (marques déposées, indemnisation, licences), aide à la collecte de fonds, infrastructure (bases de données de bogues, sites web), etc.

Transfert de droits

Tout comme pour les livres, la musique et d'autres formes de création, les logiciels impliquent une relation complexe entre les développeurs, les organisations qui distribuent leurs contributions et le grand public. Essentiellement, les contributeurs doivent prendre des mesures pour formaliser le droit d'un projet open source à utiliser et distribuer leur code.

Ainsi, de nombreuses organisations open source demandent aux contributeurs de signer un accord de licence de contributeur (*Contributor License Agreement* - CLA). Parfois, le contributeur se contente de fournir le code au projet. Le projet détient alors la propriété du code et tous les droits qui s'y rapportent, tout comme une entreprise propriétaire détient la propriété du code qu'elle a payé son personnel pour développer.

D'autres CLA laissent certains droits entre les mains des contributeurs individuels. Les contributeurs peuvent apprécier cette flexibilité, car ils peuvent continuer à contribuer au même code dans le cadre d'un autre projet ou créer leur propre entreprise autour de celui-ci.

Pour harmoniser les différentes contributions des différentes personnes, la licence attribuée au code est cruciale. Le noyau Linux est l'un des projets qui laisse la propriété entre les mains des contributeurs, de sorte qu'à l'heure actuelle, plusieurs milliers de personnes détiennent des droits sur le code Linux. Mais tous les contributeurs au code de base le placent sous la licence publique générale GNU (version 2). Ainsi, Linux est libre d'utilisation, de modification et de redistribution pour tous.

Utiliser une licence unique pour tout le code d'un projet est le moyen le plus simple de garantir qu'aucun obstacle n'empêche la distribution et l'utilisation du code. Mais parfois, un projet autorise différentes parties du code à être soumises à différentes licences, généralement parce que le projet souhaite tirer parti d'un code préexistant déjà publié sous une licence différente. Les experts en licences doivent s'assurer que les licences sont compatibles.

Règles et stratégies

De nombreuses communautés en ligne ont la réputation de permettre une liberté d'expression sans limites (une idée chevaleresque selon laquelle "tout est permis"), ce qui conduit à des insultes et à des querelles. Aujourd'hui, la plupart des communautés open source luttent contre cette tendance, qui s'empare trop facilement des gens. En revanche, les communautés modernes veulent que les gens soient constructifs, courtois, respectueux et inclusifs envers tout le monde : tous les genres, tous les groupes ethniques, tous les types de personnalité, etc.

Les attentes des membres du groupe sont généralement précisées de manière explicite dans un *code de conduite* qui explique comment interagir avec les autres participants au projet, tant en

ligne qu'en face à face. Pour être efficace, ce code de conduite doit être appliqué par le gestionnaire de communauté et les chefs de projet.

Parfois, une personne qui se moque ou dénigre de manière flagrante un autre membre de la communauté est expulsée de manière permanente ou temporaire. D'autres fois, le gestionnaire de la communauté ou un collègue annonce simplement publiquement que le comportement enfreint le code de conduite et peut discuter avec le contrevenant en dehors du forum. Souvent, le contrevenant ressentait auparavant de la frustration, un manque d'attention ou un épuisement professionnel (*burn-out*), et les membres de la communauté peuvent l'aider à trouver de meilleurs moyens de s'exprimer.

En plus du comportement social, les communautés établissent des normes de qualité. Les plus formelles consistent en des *directives de codage*, qui visent à garantir que tous les codes se ressemblent. Ces directives peuvent rappeler aux développeurs les bonnes pratiques, telles que l'utilisation de crochets autour des blocs de code, ou préciser des détails tels que la manière de nommer les variables et le type d'indentation à utiliser.

Les communautés open source ont également des règles concernant la publication de code ou d'autres livrables. Certains projets fixent des dates de publication fixes ; par exemple, Ubuntu promet une nouvelle version à support à long terme (*Long Term Support* - LTS) tous les deux ans, ainsi que des versions intermédiaires plus fréquentes. D'autres projets tiennent à jour une liste des nouvelles fonctionnalités et des corrections de bogues qu'ils souhaitent voir figurer dans la prochaine version et approuvent la publication lorsque tous les éléments de la liste ont été vérifiés. Mais contrairement à la plupart des entreprises, les communautés open source ne fixent généralement pas de délais aux participants, car on ne peut pas trop en demander aux bénévoles.

Attribution et transparence

En plus de l'accord de licence de contributeur mentionné précédemment, certains projets demandent aux contributeurs de signer un certificat d'origine du développeur (*Developer Certificate of Origin* - DCO). Dans le DCO, le contributeur s'engage à avoir le droit légal de donner le code.

Pourquoi le DCO est-il important ? Imaginez le scénario suivant : si un programmeur utilise le code d'un produit propriétaire développé par son employeur et le contribue à un projet open source, il enfreint la licence de son employeur et expose le projet open source à des risques juridiques.

Le DCO est censé s'assurer que le contributeur a bien écrit le code ou l'a obtenu légalement. Le projet open source dépend de l'honnêteté du contributeur qui remplit le certificat.

Diversité, équité, inclusion et non-discrimination

Les spécialistes en sciences sociales affirment que les projets et les entreprises tirent profit de la diversité des personnes qui les composent, qu’il s’agisse de leur sexe, de leur origine ethnique, de leur situation économique, de leur nationalité ou de leurs capacités. Toutes les organisations ont naturellement tendance à créer des liens avec des personnes qui ressemblent à leurs membres actuels. Une communauté qui valorise la diversité et l’équité doit donc consciemment former ses membres à être plus ouverts aux personnes qui ne leur ressemblent pas. Le mouvement visant à mettre en œuvre cet idéal s’appelle “diversité, équité et inclusion” (DEI).

Le code de conduite est le point de départ de la DEI. Il doit explicitement accueillir les personnes de différents genres, races, etc. Toute violation du code de conduite doit être traitée rapidement et avec une désapprobation sans ambiguïté. En effet, certaines minorités ont souffert d’exclusion et de commentaires négatifs toute leur vie, et une seule mauvaise interaction sur votre forum pourrait les amener à décider qu’elles n’ont plus aucune raison de participer. Une réponse rapide aux agressions peut les rassurer sur le fait que la communauté les soutient.

Mais la DEI va bien plus loin. La communauté doit tendre la main aux groupes qui ont besoin d’une meilleure représentation. Par exemple, si vous développez une application de recherche d’emploi, vous devez vous assurer qu’elle affiche des offres d’emploi destinées aux personnes à faibles revenus ainsi qu’aux personnes issues des classes supérieures et moyennes. Vous devez également faire la promotion de l’application auprès des communautés à faibles revenus et recruter des membres de ces communautés pour la tester.

Votre communauté pourrait tirer profit de la recherche d’organisations qui forment des personnes issues de communautés marginalisées et auprès desquelles vous pourriez recruter de nouveaux membres pour votre équipe. Bon nombre de ces organisations sont implantées au niveau local et ne sont pas très connues à l’échelle nationale ou internationale.

Il est essentiel de comprendre les besoins des communautés marginalisées. Votre site web est-il accessible aux personnes malvoyantes ? Traduisez-vous votre documentation dans des langues familières à la communauté cible ? Vous devriez peut-être créer des forums spécifiques dans d’autres langues.

La plupart des communications au sein des communautés open source sont asynchrones et en ligne. Mais si vous organisez des réunions ou des sessions de chat synchronisées, pensez à la situation géographique des participants et trouvez des moyens d’inclure tout le monde. Par exemple, lorsque des personnes de nombreux pays et régions communiquent en anglais, essayez d’utiliser un vocabulaire et une grammaire simples.

Enfin, si votre équipe compte des membres issus d’une minorité, veillez à les écouter. L’un des

symptômes bien connus de l'exclusion consiste à ignorer ce que disent les minorités ou à sous-estimer son importance.

D'autre part, ne chargez pas les membres issus de minorités d'expliquer les besoins de leur communauté : tout le monde devrait être chargé de cette recherche. Les membres issus de minorités peuvent vous apporter une aide précieuse en matière de sensibilisation, mais ne les mettez pas sous pression pour qu'ils le fassent. Chaque personne devrait avoir les mêmes chances de participer comme elle le souhaite, sans avoir à jouer le rôle de minorité symbolique.

Exercices guidés

1. Pourquoi tous les contributeurs ne donnent-ils pas leur code au projet et ne renoncent-ils pas à tous leurs droits sur ce code ?

2. Si quelqu'un souhaite contribuer au projet mais ne sait pas programmer, comment peut-il aider ?

3. Pourquoi de nombreux projets open source rejoignent-ils une fondation ?

Exercices d'exploration

1. Vous êtes un commiteur sur votre projet. Quelqu'un soumet du code qui provient d'un autre projet (mais le contributeur en détient les droits). Le code est formaté de manière complètement différente du reste de votre code. Que faites-vous ?

2. Vous avez créé un logiciel propriétaire et souhaitez intégrer certaines parties de celui-ci dans un projet open source. Dans quelles circonstances pouvez-vous continuer à proposer votre produit propriétaire ?

3. Deux personnes de votre liste de diffusion commencent à se disputer au sujet d'une fonctionnalité de votre code. La dispute s'envenime progressivement jusqu'à ce que l'une des deux personnes traite l'autre d'idiot. Comment pouvez-vous gérer cette situation ?

Résumé

Dans cette leçon, vous avez découvert ce que signifie faire partie d'une communauté et comment les communautés restent productives. Vous avez découvert différents types de contributions, de contributeurs et de rôles. Vous avez découvert comment les communautés contrôlent le droit d'utiliser les contributions. Vous avez découvert les règles d'une communauté et comment elles protègent tout le monde, y compris les personnes de races et de sexes différents, contre les agressions verbales.

Réponses aux exercices guidés

1. Pourquoi tous les contributeurs ne donnent-ils pas leur code au projet et ne renoncent-ils pas à tous leurs droits sur ce code ?

Le contributeur peut souhaiter contribuer avec le même code dans un autre projet ou commercialiser un produit basé sur ce code, et souhaite donc conserver certains droits.

2. Si quelqu'un souhaite contribuer au projet mais ne sait pas programmer, comment peut-il aider ?

Les contributeurs peuvent avoir de nombreux rôles autres que le codage. Parmi ceux-ci, on peut citer la documentation, les tests et le signalement de bogues, la gestion de la communauté, la participation à des forums, la promotion du projet et la création d'illustrations.

3. Pourquoi de nombreux projets open source rejoignent-ils une fondation ?

Une fondation s'occupe de nombreuses tâches juridiques, financières et autres que la communauté du projet pourrait avoir du mal à accomplir.

Réponses aux exercices d'exploration

1. Vous êtes un commiteur sur votre projet. Quelqu'un soumet du code qui provient d'un autre projet (mais le contributeur en détient les droits). Le code est formaté de manière complètement différente du reste de votre code. Que faites-vous ?

Les normes de codage de votre projet doivent expliquer clairement comment formater le code. Remerciez le contributeur, indiquez-lui les normes à respecter et demandez-lui de reformater le code. Il existe parfois des outils automatisés permettant de reformater le code comme vous le souhaitez. Si le contributeur n'a pas le temps de procéder au reformatage, cherchez un membre junior de votre équipe qui puisse s'en charger.

2. Vous avez créé un logiciel propriétaire et souhaitez intégrer certaines parties de celui-ci dans un projet open source. Dans quelles circonstances pouvez-vous continuer à proposer votre produit propriétaire ?

La réponse dépend du contrat de licence de contributeur. Si le CLA vous oblige à céder le code au projet, en lui accordant tous les droits, vous ne pourrez peut-être plus continuer à proposer votre produit propriétaire. Cependant, s'il vous permet de conserver vos droits sur le code, vous pouvez le publier sous la licence de votre choix dans votre produit propriétaire.

3. Deux personnes de votre liste de diffusion commencent à se disputer au sujet d'une fonctionnalité de votre code. La dispute s'envenime progressivement jusqu'à ce que l'une des deux personnes traite l'autre d'idiot. Comment pouvez-vous gérer cette situation ?

Toute personne qui remarque l'escalade et le commentaire abusif doit réagir aussi rapidement que possible. Le gestionnaire de communauté est en fin de compte responsable de réparer les dommages. La personne qui intervient doit publier un commentaire à l'ensemble de la liste indiquant que le comportement enfreint le code de conduite du projet (qui, espérons-le, exclut un tel comportement). La personne qui intervient peut également contacter séparément les personnes impliquées afin de s'assurer qu'elles sont satisfaites de la résolution du conflit et qu'elles comprennent comment discuter de manière constructive en cas de désaccord.



Thème 056: Collaboration et communication



**Linux
Professional
Institute**

056.1 Outils de développement

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 056.1

Valeur

2

Domaines de connaissance les plus importants

- Comprendre les principaux outils de développement de logiciels
- Comprendre les principaux environnements de déploiement
- Comprendre les principaux types de tests de logiciels
- Comprendre les concepts d'intégration et de livraison continues (CI/CD)

Liste partielle de termes, fichiers et utilitaires utilisés

- Environnements de développement intégré (IDE)
- Linters
- Compilateurs
- Débogueurs
- Rétro-ingénierie
- Réusinage de code (Refactoring)
- Tests unitaires
- Tests d'intégration
- Tests d'acceptation
- Tests de performance

- Tests fumigatoire
- Tests de régression
- Systèmes de production, de préproduction et de développement
- Systèmes de développement locaux
- Systèmes de développement à distance
- Pipelines CI/CD



Linux
Professional
Institute

Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	056 Collaboration et communication
Objectif :	056.1 Outils de développement
Leçon :	1 sur 1

Introduction

Il existe des milliers d'outils de développement logiciel, open source ou propriétaires. Et pourquoi pas ? Les programmeurs adorent développer des outils pour eux-mêmes et leurs collègues ; il est donc tout à fait naturel qu'ils investissent beaucoup de temps à rechercher un outil plus performant, qui élimine certaines sources d'irritation dans leur flux de travail ou facilite le déploiement.

Cette leçon se concentre sur le *processus* de développement et explique comment différents types d'outils de développement s'y intègrent. Peu d'outils spécifiques seront cités, car n'importe lequel d'entre eux pourrait être considéré comme obsolète ou dépassé, et être remplacé par un nouvel outil préféré au moment où vous lisez cette leçon.

Objectifs de développement

Les outils de programmation jonglent avec plusieurs objectifs qui sont parfois contradictoires. Voici quelques exemples d'objectifs de développement :

- Produire des programmes robustes et précis.

- Produire des programmes qui s'exécutent rapidement.
- Produire des programmes qui évoluent facilement.
- Produire des programmes hautement adaptables à différents types d'utilisateurs, différents appareils (tels que les ordinateurs portables, les téléphones cellulaires et les tablettes) et différents environnements.
- Accélérer le processus de développement.
- Accélérer le déploiement des nouvelles fonctionnalités ou des corrections de bogues.
- Réduire les tâches fastidieuses, ainsi que le nombre d'erreurs de programmation écrites qui se glissent dans les phases de test du programme.
- Supporter le code et les appareils anciens que la société a du mal à remplacer.
- Collaborer avec d'autres outils familiers.
- Permettre une restauration facile en cas d'erreurs ou de changements dans les plans.

Ces objectifs, entre autres, ont conduit aux processus décrits dans cette leçon et aux outils de développement qui en ont résulté.

Processus généraux de développement

Cette section compare deux modèles généraux d'importance historique : le *modèle en cascade* (présenté dans une leçon précédente) et *l'intégration continue/la livraison continue* (*Continuous Integration/Continuous Delivery* - CI/CD).

Modèle en cascade

Bien que le modèle en cascade soit encore largement utilisé, en particulier par les grandes organisations, il a perdu la faveur de nombreux développeurs. Ce modèle était populaire entre les années 1950 et 1970. Certains éléments de ce modèle sont encore largement utilisés aujourd'hui, tels que les définitions formelles des exigences et les programmes de test juste avant leur publication (*assurance qualité*).

Même à l'époque où le terme "cascade" a été inventé pour désigner ce modèle, celui-ci était déjà largement discrédité. Parmi ses problèmes, on peut citer :

- Il était difficile de modifier les exigences une fois le développement commencé.
- Les exigences et les conceptions étaient souvent mal comprises, car un texte rédigé dans un langage simple peut être ambigu. Ce problème a conduit à la création de produits qui ne répondaient pas aux exigences prévues et dont la correction a pris des mois.

- Le processus était lent. Une nouvelle version ne pouvait être publiée qu’une fois par an, voire moins souvent.
- Les bogues causés par l’interaction entre différents modules étaient difficiles à détecter jusqu’à la fin du processus, ce qui entraînait des retards supplémentaires.
- Ce processus a créé des barrières entre les différentes parties de l’organisation, ce qui a nui à la qualité du produit ainsi qu’à *l’esprit de corps* au sein de l’organisation.

Principes de l’intégration continue/livraison continue

Le modèle en cascade a été remis en question dans les années 1970 par une série de mouvements qui ont conduit au modèle le plus célèbre (sinon universellement utilisé) aujourd’hui : CI/CD (*Continuous Integration/Continuous Delivery*). Les étapes de l’adoption des pratiques CI/CD comprennent le *Manifeste Agile*, publié en 2001, SCRUM et DevOps. Le nouveau modèle repose sur les principes d’une communication étroite entre les membres de l’équipe, de l’implication de l’utilisateur final ou du client, du déploiement rapide des corrections de bogues et des nouvelles fonctionnalités, et de tests rigoureux et continus afin de préserver la qualité dans un environnement en constante évolution — parfois même chaotique.

Le CI/CD automatise autant d’étapes que possible dans les phases allant de l’écriture du code au déploiement du programme finalisé auprès des utilisateurs finaux. Le CI/CD nécessite de définir chaque étape de manière formelle et de remplacer une action humaine (telle que l’installation manuelle d’un logiciel) par une procédure exécutée par un programme. Ainsi, le CI/CD s’inscrit dans un mouvement connu sous les expressions “tout sous forme de code” (*everything as code*) et “infrastructure sous forme de code” (*infrastructure as code*).

De plus, une fois qu’une équipe a intégré ses processus dans le code, ces processus peuvent être corrigés, mis à niveau et enregistrés dans l’historique, tout comme le code. Tout ce qui est écrit par l’équipe, qu’il s’agisse de programmes, de procédures automatisées ou de documentation, doit être stocké dans un système de contrôle de version, qui sera décrit dans une prochaine leçon.

Lorsque plusieurs procédures sont automatisées, elles peuvent s’exécuter de manière séquentielle. Ainsi, les équipes parlent souvent d’un *pipeline CI/CD*, ce qui signifie qu’une étape réussie dans le pipeline déclenche automatiquement un ou plusieurs processus ultérieurs. Un pipeline doit être surveillé de manière automatisée afin que toute défaillance en cours de route entraîne l’arrêt du pipeline et la notification des membres de l’équipe concernés.

Bien que les sections suivantes traitent séparément de la CI et de la CD, celles-ci ont tendance à être combinées et la frontière entre elles est floue.

Intégration continue

L'intégration continue (*Continuous Integration* - CI) désigne l'incorporation rapide de petites modifications dans le code. Le référentiel principal utilisé pour créer le produit destiné aux utilisateurs finaux est appelé *référentiel central* (*core repository* ou *core repo*). Chaque programmeur crée un espace de travail personnel (souvent appelé *sandbox*) sur son ordinateur et extrait du référentiel central les fichiers dont il a besoin pour effectuer des corrections ou des mises à niveau.

Le programmeur peut utiliser un ordinateur portable ou de bureau personnel (appelé *système de développement local*), télécharger les parties pertinentes du référentiel central, puis téléverser les modifications après les avoir testées localement. Il peut également tirer parti de la tendance populaire appelée "cloud computing" et travailler sur un système partagé géré par son organisation, appelé *système de développement distant*.

Le programmeur vérifie généralement les erreurs à l'aide d'un *débogueur*, un programme distinct qui exécute le travail du programmeur dans un environnement contrôlé. Nous examinerons plus tard les débogueurs et autres outils permettant de détecter les erreurs.

Afin de détecter les erreurs le plus tôt possible dans le processus de développement, le programmeur effectue également des tests sur le code avant de le téléverser dans le référentiel central. Ces tests sont appelés *tests unitaires* car chacun d'entre eux se concentre sur un élément minuscule du code : par exemple, une fonction a-t-elle incrémenté un compteur comme prévu ?

La dernière étape du CI consiste à vérifier les modifications apportées par le programmeur dans le référentiel central. À ce stade, des outils exécutent des *tests d'intégration* afin de s'assurer que le programmeur n'a rien endommagé dans l'ensemble du projet.

Peu importe les progrès réalisés en matière d'automatisation, un membre expert de l'équipe doit intervenir au moment de l'intégration afin de s'assurer que le changement est souhaité par l'équipe. Les tests automatisés permettent de vérifier que rien n'a été endommagé et même si le changement produit l'effet souhaité dans le programme. Cependant, ces tests ne peuvent pas vérifier tous les principes jugés importants par l'équipe.

Par conséquent, avant de procéder au déploiement, un membre de l'équipe doit vérifier la sécurité, le respect des normes de codage, la documentation appropriée et d'autres principes. (Il existe bien sûr des outils pour certaines de ces tâches ; nous les examinerons plus loin dans cette leçon).

De nombreux tests sont effectués pendant l'intégration continue, et ceux-ci doivent être rapides et fiables afin de suivre le rythme du développement moderne. C'est pourquoi les outils modernes d'intégration de code et d'exécution de tests sont automatisés. Un programmeur doit pouvoir

exécuter toute une série de tests unitaires à l'aide d'une seule commande ou d'un simple clic sur un bouton.

En général, un test d'intégration partiel ou complet est exécuté chaque fois que le programmeur téléverse du code dans le référentiel central. Toute erreur détectée par les tests peut être rapidement signalée au programmeur.

Livraison continue

Comme expliqué dans la section précédente, l'intégration continue consiste en des procédures automatisées qui aboutissent à une nouvelle version du programme dans le répertoire central. La *livraison continue* (*Continuous Delivery* - CD) désigne tout ce qui se passe après cette étape pour mettre la nouvelle version à la disposition des utilisateurs finaux. Le D dans CD est parfois remplacé par "développement" (*development*) ou "déploiement" (*deployment*) en plus de "livraison" (*delivery*).

Les principales tâches du CD consistent à tester minutieusement le code et à le téléverser sur les ordinateurs des utilisateurs ou dans un référentiel d'applications.

La phase CD exécute une série de tests afin de garantir au maximum le bon fonctionnement du produit. Un ensemble plus large de tests d'intégration peut être exécuté, ainsi qu'un certain nombre d'autres tests visant à déterminer les impacts sur les utilisateurs, les performances et la sécurité. Une section ultérieure de cette leçon décrit certains de ces tests.

Les tests doivent être effectués sur des systèmes différents des systèmes de production qui servent les utilisateurs. Non seulement une faille catastrophique pourrait mettre hors service un système de production, mais elle pourrait également corrompre les données des utilisateurs. De plus, les tests occupent du temps système et dégradent les performances pour les utilisateurs.

Pour les tests, il est donc préférable de mettre en place un environnement informatique complet qui reflète votre environnement de production, avec du matériel et des logiciels similaires. L'environnement intermédiaire dans lequel les tests sont effectués avant le déploiement est souvent appelé environnement de *simulation* (*staging environment*).

Bien sûr, il ne serait pas pratique de créer un environnement de test de la même taille que l'environnement de production, mais les éléments essentiels de l'environnement de production (tels que les bases de données) devraient être inclus.

L'automatisation des CD exige notamment de distinguer les environnements de test et de production. Toutes les installations et tous les processus liés au code doivent être adaptés à l'environnement cible.

La livraison continue offre le plus grand potentiel de développement rapide lorsque le code est un service fonctionnant sur les propres systèmes de l'organisation. Si vous êtes un site de vente au détail, par exemple, proposant une page web interactive, vous avez accès à tous vos serveurs web. Vous pouvez les mettre à jour plusieurs fois par jour, si vous le souhaitez, et revenir rapidement en arrière si les modifications produisent des problèmes pour les utilisateurs.

Outils usuels de développement logiciel

Ces sections présentent les types d'outils couramment utilisés par les équipes de programmation à trois niveaux : génération de code, tests et déploiement.

Compilateurs

Un outil de programmation fondamental est le *compilateur*, qui transforme des langages de programmation très sophistiqués et de haut niveau en instructions exécutables par le processeur de l'ordinateur.

Les ordinateurs exécutent du *code machine*, qui consiste en des chaînes de bits (des uns et des zéros) que le processeur transforme en instructions et en données : chargement d'une valeur depuis la mémoire vers un registre, addition des valeurs de deux registres, etc. Les processeurs se distinguent par leurs différents formats et ensembles d'instructions, ils ont donc également des codes machine différents. Les fabricants s'efforcent de concevoir de nouveaux processeurs qui utilisent le même code machine afin de permettre aux clients d'exécuter leurs anciens programmes sur les nouveaux processeurs. Lorsque les fabricants procèdent ainsi, on parle de *familles* de processeurs.

Dans les premières années de la programmation, l'étape suivante a été le *langage assembleur*, qui fournissait des termes lisibles par l'homme, tels que ADD, pour chaque instruction. Les programmeurs écrivaient en langage assembleur et soumettaient leur code à un outil appelé *assembleur* afin de traduire le langage assembleur en code machine. Le code machine est également appelé langage machine.

Ensuite, des langages de plus haut niveau ont été créés. Aujourd'hui, il est possible de créer des flux de contrôle complexes sans même en spécifier les détails ; il suffit d'indiquer les résultats souhaités. Ces programmes nécessitent un compilateur pour transformer le code source en code machine. Les compilateurs peuvent effectuer des transformations et des optimisations assez intelligentes.

De nombreux langages disposent de plusieurs compilateurs. Vous trouverez un choix de compilateurs pour les langages très populaires, tels que C et Java. Dans la communauté libre et open source, la plupart des gens utilisent soit la GNU Compiler Collection (GCC), soit le

compilateur LLVM pour C et C++.

À l'origine, un compilateur compilait chaque fichier de code source pour produire un *fichier objet* intermédiaire, puis combinait tous les fichiers objets en un seul programme en invoquant un autre outil appelé *éditeur de liens* (*linker*). Les compilateurs modernes peuvent compiler plusieurs fichiers à la fois afin d'effectuer des optimisations qui dépassent les limites des fichiers.

Les compilateurs étaient utilisés pour générer un langage assembleur, ce qui pouvait être utile, car chaque type de processeur utilise un code machine différent, mais peut utiliser le même langage assembleur. Il existe plusieurs variantes de langages assembleurs. La dernière étape de la compilation et de l'édition de liens consistait à produire du code machine. Comme pour l'édition des liens, les compilateurs modernes savent comment assembler le code en code machine.

Mais il existe une autre étape dans de nombreux langages modernes : le code intermédiaire, généralement appelé *code en bytes* (*byte code*). Ce code a été compilé dans un format binaire suffisamment haut niveau pour être portable. Par exemple, le langage Java est compilé en byte code afin que le programme puisse être chargé sur de nombreux types de processeurs différents.

En produisant du byte code à partir du code source, le compilateur a accompli une grande partie du travail. Chaque ordinateur héberge alors sa propre version d'un programme appelé *machine virtuelle* qui achève la transformation du byte code en un ensemble d'instructions que la machine virtuelle peut exécuter. Parfois, le byte code est également compilé en code machine. Passer du byte code à un ensemble d'instructions est plus pratique pour les utilisateurs finaux que passer d'un langage de programmation de haut niveau au code machine. Cela a constitué un avantage considérable pour Java lors de sa création, car ses concepteurs souhaitaient qu'il fonctionne dans un plug-in de machine virtuelle pour les navigateurs web, où le processeur et l'environnement d'exploitation de l'utilisateur pouvaient être très variés.

Les concepteurs de Java ont vanté les avantages du byte code à travers le slogan marketing "Compiler une fois, exécuter partout". D'autres avantages du byte code sont apparus par la suite. De nouveaux langages ont pu être créés, offrant une expérience très différente aux programmeurs (dans l'espoir de faciliter le travail de programmation et de produire un code plus facile à maintenir) tout en créant le même byte code pris en charge par les machines virtuelles Java. Les fonctions de ces langages étaient simples à intégrer dans les programmes Java existants.

Enfin, il existe certains langages, tels que Python, pour lesquels vous n'avez pas besoin de compiler le code source : il vous suffit d'entrer des instructions dans un outil de traitement appelé *interpréteur*, qui convertit directement le code en code machine et l'exécute. Les interpréteurs sont plus lents que les compilateurs, mais certains sont devenus suffisamment efficaces pour ne pas nuire beaucoup aux performances. Néanmoins, certaines bibliothèques populaires en langage Python comprennent des fonctions accessibles aux développeurs dans le langage interprété, mais

programmées en langage C afin d'accélérer l'exécution.

De nombreux langages interprétés fournissent également des compilateurs. Ceux-ci produisent soit du byte code, soit du code machine, qui s'exécute ensuite plus rapidement que l'interpréteur.

Il existe des outils de production qui aident les programmeurs à gérer les fichiers et les fonctions. Un programme complexe peut contenir des centaines de fichiers, et le programmeur devra compiler différentes combinaisons de fichiers en utilisant différentes options de compilation à différents moments (par exemple, pour faciliter le débogage). Un outil de production permet aux programmeurs de stocker différentes options et combinaisons de fichiers, et de choisir facilement le type de production souhaité. Maven et Gradle sont des outils de production courants pour Java et les langages apparentés.

Outils de génération de code

Le programmeur n'a plus besoin de commencer à coder à partir d'un écran vide. Depuis peu, il existe des services qui génèrent du code à partir d'une description en texte brut de ce que vous souhaitez obtenir. Il s'agit d'une forme d'IA générative et, comme pour d'autres services de ce type, certaines personnes se plaignent qu'elle s'inspire du travail d'anciens programmeurs sans les rémunérer et qu'elle produit un code source moins robuste (pour l'instant). Au-delà des controverses éthiques, de nombreux programmeurs affirment que la génération automatique de code a considérablement amélioré leur productivité.

Une forme de génération de code disponible depuis de nombreuses années dans plusieurs langages de programmation est le *refactoring*. Il examine un programme volumineux, qui peut facilement évoluer au fil du temps en un enchevêtrement de fichiers et de fonctions. Le *refactoring* déplace les fonctions afin de créer une structure plus logique pour le programme, dans le but d'améliorer la maintenabilité et de réduire la duplication du code source.

Certains programmeurs sont chargés de reproduire le fonctionnement d'autres codes. Ils peuvent être amenés à écrire un nouveau programme pour remplacer un ancien programme dont le code source est manquant et qui doit être retiré. Ils peuvent également être amenés à imiter le programme d'un concurrent. Ce type de recherche est appelé *ingénierie inverse*. Un outil utile à cette fin est le *désassembleur*, qui transforme le code machine en langage assembleur. Il existe également des désassembleurs pour le byte code.

Débogueurs

La plupart des programmeurs passent plus de temps à déboguer qu'à coder. Les gens ne pensent tout simplement pas de manière parfaitement logique et oublient donc certains détails dont l'ordinateur a besoin pour exécuter le programme comme vous le souhaitez. Ainsi, votre code est

susceptible d'échouer lors de la première tentative, et vous pouvez tirer un grand profit d'un *débogueur* pour détecter l'erreur.

Un débogueur facilite les efforts de recherche intensifs qui peuvent réduire considérablement le temps nécessaire à la recherche d'erreurs.

Un programmeur peut demander au programme de s'arrêter à un endroit clé du programme (un *point d'arrêt* ou *breakpoint*), tel que le début d'une fonction ou d'une boucle. Le débogueur peut afficher les valeurs des variables et même des registres de l'ordinateur à l'endroit actuel du programme. Le programmeur peut également exécuter chaque instruction, une à la fois, et voir les résultats (*pas à pas* ou *single stepping*). Si le programmeur souhaite voir quand et comment une variable change pendant l'exécution, il peut définir un *point de surveillance* (*watchpoint*).

Le débogueur le plus connu dans le monde du logiciel libre et open source, en particulier pour C et C++, est le GNU debugger, qui fonctionne avec le compilateur GNU mentionné précédemment. D'autres langages disposent également de débogueurs dédiés.

Outils analytiques

Bien que le débogage permette généralement de détecter assez rapidement les bogues, il est préférable d'éliminer les fautes de frappe et autres problèmes élémentaires dès le début du processus de programmation. Il existe de nombreux outils analytiques ingénieux pour vérifier un programme. L'*analyse statique* examine le code d'un programme. L'*analyse dynamique* exécute un programme et vérifie s'il y a des problèmes pendant son exécution.

L'une des premières formes d'analyse statique était appelée *linter*. Elle peut détecter, par exemple, si vous attribuez la valeur d'une variable à virgule flottante à un entier. Cela peut ou non produire des problèmes, et peut ou non être détecté par le compilateur. En production, cela peut conduire à des résultats incorrects.

De nos jours, la plupart des compilateurs peuvent faire le travail d'un linter. Certains compilateurs, comme celui du langage Rust, sont réputés pour leur rigueur dans le rejet des codes mal écrits.

De nombreux autres types d'outils analytiques fonctionnent indépendamment du compilateur. Par exemple, les outils d'analyse de sécurité peuvent détecter les problèmes qui rendent un programme vulnérable au piratage. Une erreur courante chez les développeurs consiste, par exemple, à appeler une fonction dans leur code sans vérifier si cette fonction a renvoyé une erreur.

Environnements de développement intégrés (EDI)

De nombreux programmeurs utilisent des éditeurs de texte pour saisir et modifier leur code. Les éditeurs de texte diffèrent des traitements de texte, qui introduisent de nombreuses mises en forme (italique, gras, puces, listes numérotées, etc.). Le traitement de texte produit un texte sans fioritures, comme l'exigent les langages de programmation.

Il existe également des outils sophistiqués destinés à aider les programmeurs à développer leurs programmes. Ces outils sont sensibles au langage de programmation utilisé. Par exemple, si vous commencez à taper le nom d'une variable ou d'une fonction, l'outil peut vous suggérer une complétion. Ces outils peuvent vérifier les erreurs pendant que vous codez, formater le code de manière cohérente et agréable, exécuter des outils d'analyse et des débogueurs, compiler le code, gérer les enregistrements dans les systèmes de contrôle de version et s'occuper d'autres tâches à votre place. C'est pourquoi on les appelle des *Environnements de Développement Intégrés* (EDI ou IDE pour *Integrated Development Environments*).

Eclipse est un IDE open source populaire.

Types de test logiciels courants

Les tests constituent une partie importante du développement logiciel. Les programmeurs exécutent des tests unitaires au fur et à mesure qu'ils développent le code. D'autres types de tests sont généralement exécutés lorsqu'un programmeur réintègre le code dans le référentiel central ou pendant le déploiement.

Tests unitaires

Nous avons vu qu'un programmeur doit prendre grand soin de détecter les erreurs avant de soumettre du code pour intégration dans le référentiel central. Les tests unitaires sont essentiels pour détecter les erreurs.

L'écriture de ces tests est à la fois un art et une science, et le volume du code de test peut dépasser celui du code de production. Il existe même un modèle de développement appelé *développement piloté par les tests* (*Test-Driven Development* - TDD), dans lequel les programmeurs écrivent les tests avant d'écrire le code qu'ils souhaitent tester. Les partisans du TDD affirment qu'il comble les lacunes des tests et permet de s'assurer que le code fait ce que le programmeur souhaite qu'il fasse.

Il est important de tester les éléments qui ne fonctionnent pas correctement pendant l'exécution du programme, ainsi que ceux qui fonctionnent correctement. Si l'utilisateur, ou une autre partie du programme, soumet une entrée non valide à une fonction, il est important que la fonction

détecte le problème et affiche un message d'erreur approprié.

JUnit est un outil open source populaire permettant d'exécuter des tests unitaires sur des programmes Java.

Intégration, régression et tests fumigatoires

Alors que les tests unitaires se concentrent sur les actions individuelles de fonctions particulières, l'équipe doit également effectuer des tests à un niveau supérieur afin de s'assurer que le produit, dans son ensemble, fonctionne correctement. Les tests imitent généralement le comportement des utilisateurs. Par exemple, dans une application de gestion de restaurant, les tests pourraient vérifier si l'utilisateur obtient l'article qu'il a demandé.

Lorsqu'une modification apportée à un programme rend inopérante une fonction qui fonctionnait auparavant, cette défaillance est appelée *régression* et les tests sont appelés *tests de régression*.

Lorsqu'une équipe prépare un produit pour sa livraison, la première phase de test est souvent très courte. L'équipe vérifie les activités les plus importantes effectuées par un programme et arrête les tests si des erreurs surviennent, ce qui permet de gagner du temps. Ce type de test est appelé *test fumigatoire* (*smoke test*), car une application qui fonctionne si mal est comparable à un appareil défectueux qui prend feu.

Certains produits impliquent une interaction avec l'utilisateur ; les applications web et mobiles en sont des exemples courants. C'est pourquoi des outils ont été créés pour émuler les interactions des utilisateurs. Le test s'exécute automatiquement, déclenchant le code qui aurait été exécuté si un utilisateur avait appuyé sur le bouton. Selenium est un outil populaire dans cette catégorie.

Tests d'acceptation

Les programmes dotés d'une interface utilisateur nécessitent un niveau de test supplémentaire, au-delà de la simple vérification de leur bonne réaction à certaines entrées. Les programmes doivent également s'afficher correctement à l'écran. Les *tests d'acceptation* vérifient l'impact du programme sur l'utilisateur. Les équipes d'assurance qualité effectuent généralement ces tests après que les tests d'intégration et de régression ont démontré que le programme est formellement correct et répond aux attentes des utilisateurs.

Tests de sécurité

La sécurité est évidemment essentielle, et même une petite faille peut exposer une organisation à des dommages importants. Nous avons vu que les programmeurs peuvent utiliser des outils analytiques pour vérifier la sécurité du code. Au stade de l'assurance qualité, des tests permettent

également de déterminer si le programme présente des vulnérabilités. Les tests exécutent le programme avec des entrées malveillantes et s'assurent que le programme rejette l'entrée sans échouer, sans effectuer d'actions non souhaitées et sans révéler d'informations sensibles.

Un type de test qui s'est avéré utile dans certaines situations est le test à données aléatoires (*fuzz testing*). La plateforme de test génère simplement des chaînes de caractères aléatoires et les soumet comme entrée au programme. Cela peut sembler être une perte de temps, mais cela permet souvent de détecter des vulnérabilités que les tests ordinaires ne permettent pas de détecter.

Tests de performance

Une fois que d'autres tests ont permis de vérifier que le programme fonctionne correctement, les équipes doivent déterminer s'il est suffisamment rapide. Les *tests de performance* nécessitent un environnement similaire à celui dans lequel les utilisateurs interagiront avec le programme. Par exemple, si les utilisateurs soumettent des requêtes à distance via un réseau, les tests de performance doivent également être effectués sur un réseau longue distance.

Certaines bibliothèques de programmation sont testées à l'aide de *benchmarks* : des tests standard utilisés pour comparer différentes bibliothèques ou différentes versions d'une même bibliothèque.

Environnements de déploiement courants

Un outil CI/CD permet de construire des pipelines de manière sophistiquée. À l'instar des programmes informatiques, un pipeline peut contenir des tests et des branches. Grâce aux branches, vous pouvez effectuer un ensemble d'activités dans l'environnement de test et un autre dans l'environnement de production. Vous pouvez utiliser l'outil pour installer automatiquement la base de données appropriée ou tout autre logiciel nécessaire à différents programmes.

La CD recoupe le DevOps. Dans les environnements cloud, les outils CD et DevOps créent de manière automatisée les systèmes informatiques virtuels avec tous les composants nécessaires au fonctionnement des programmes. Des outils automatisés (parfois appelés outils d'*orchestration*) vérifient les défaillances des systèmes virtuels et les redémarrent automatiquement.

La principale fonction de l'outil CD est de lancer et de parcourir chaque pipeline. L'outil vérifie les résultats de chaque étape du pipeline et choisit de poursuivre ou d'arrêter. L'outil permet également la planification et journalise ses activités.

Il arrive souvent que vous deviez exécuter une tâche à plusieurs reprises avec des variations mineures. Par exemple, les équipes font la distinction entre le déploiement dans un

environnement de test et le déploiement en production. C'est pourquoi les outils CD fournissent des paramètres que vous pouvez remplir avec différentes valeurs lorsque vous exécutez le pipeline.

Parfois, un processus nécessite des commandes qui étaient traditionnellement saisies au niveau du terminal. Ainsi, un outil CD fournit des mécanismes permettant d'exécuter des commandes arbitraires. Généralement, il fournit des crochets (*hooks*) tels que `preprocess` pour exécuter des commandes avant une étape du pipeline et `postprocess` pour exécuter des commandes après une étape du pipeline.

Jenkins est probablement l'outil open source le plus populaire pour l'orchestration décrite dans cette section.

Exercices guidés

1. Quels sont les moyens de vérifier la sécurité d'un programme ?

2. Pourquoi écrivez-vous plusieurs tests unitaires pour une seule fonction du programme ?

Exercices d'exploration

1. Votre équipe a hérité d'une ancienne application qui fonctionne lentement et à laquelle il est difficile d'ajouter des fonctionnalités. Comment améliorer cette application sans la supprimer et en créer une nouvelle à partir de zéro ?

2. Dans les grands projets, il arrive souvent que l'équipe A souhaite qu'une fonctionnalité soit implémentée dans une partie du système gérée par l'équipe B, mais que l'équipe B ne considère pas cette fonctionnalité comme une priorité. Comment l'équipe A peut-elle coder cette fonctionnalité dans le cadre du projet de l'équipe B ?

Résumé

Cette leçon a présenté les types d'outils utilisés tout au long du développement : compilateurs et autres outils de génération de code, analyseurs, tests et outils CI/CD qui automatisent l'intégration et la livraison. Il existe de nombreuses options pour chacune de ces activités, et un outil populaire aujourd'hui peut être remplacé dans un an. Comprendre comment ces outils s'intègrent dans le processus de développement vous aide à identifier ce dont vous avez besoin.

Réponses aux exercices guidés

1. Quels sont les moyens de vérifier la sécurité d'un programme ?

Tout d'abord, des experts humains peuvent examiner le code.

Il existe de nombreux outils d'analyse statique et dynamique permettant de détecter les mauvaises pratiques de programmation qui exposent un programme à des menaces de sécurité.

D'autres outils soumettent des entrées malveillantes à des programmes en cours d'exécution et vérifient leurs réactions.

2. Pourquoi écrivez-vous plusieurs tests unitaires pour une seule fonction du programme ?

Une fonction de programme doit généralement être exécutée avec de nombreuses variantes d'entrée, et chaque variante mérite son propre test. Par exemple, la fonction peut traiter une valeur d'entrée nulle d'une manière particulière. Vous devez également anticiper les entrées non valides et écrire des tests pour montrer que la fonction les traite correctement.

Réponses aux exercices d'exploration

1. Votre équipe a hérité d'une ancienne application qui fonctionne lentement et à laquelle il est difficile d'ajouter des fonctionnalités. Comment améliorer cette application sans la supprimer et en créer une nouvelle à partir de zéro ?

Tout d'abord, assurez-vous que le projet est sous contrôle de version, s'il ne l'était pas déjà.

Après avoir ajouté une nouvelle fonctionnalité, effectuez des tests de régression pour déterminer où le programme échoue, puis chargez des programmeurs de déterminer quelles fonctions sont responsables. Ces fonctions peuvent être remplacées de manière sélective.

Les tests de performance permettent d'identifier les fonctions particulières qui s'exécutent lentement, ce qui vous permet de concentrer vos efforts sur la correction ou le remplacement du code le moins efficace.

Si le code est écrit dans un langage qui n'est plus populaire, envisagez d'ajouter des fonctionnalités dans un langage préféré par l'équipe. Assurez-vous que les fonctions du nouveau langage peuvent être intégrées aux anciennes fonctions.

2. Dans les grands projets, il arrive souvent que l'équipe A souhaite qu'une fonctionnalité soit implémentée dans une partie du système gérée par l'équipe B, mais que l'équipe B ne considère pas cette fonctionnalité comme une priorité. Comment l'équipe A peut-elle coder cette fonctionnalité dans le cadre du projet de l'équipe B ?

L'équipe B peut autoriser l'équipe A à créer une nouvelle branche, à coder la fonctionnalité et à soumettre la branche à l'équipe B afin qu'elle soit intégrée à son projet. Cependant, l'équipe A ne doit pas être autorisée à faire tout ce qu'elle veut. L'équipe B doit fournir à l'équipe A la documentation et l'aide nécessaires pour qu'elle respecte les normes du projet. Un membre de l'équipe B doit également examiner la soumission de l'équipe A et effectuer tous les tests d'intégration habituels.

Cette forme de collaboration est parfois appelée *InnerSource*, car elle ressemble à l'open source, mais se déroule au sein d'une seule organisation.



**Linux
Professional
Institute**

056.2 Gestion du code source

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 056.2

Valeur

3

Domaines de connaissance les plus importants

- Comprendre les dépôts de code source (publics et privés)
- Comprendre les principes de la gestion du code source et de l'organisation des dépôts
- Sensibilisation aux systèmes SCM couramment utilisés (Git, Subversion, CVS)
- Sensibilisation aux termes Système de gestion de versions (VCS), Système de gestion de révisions et Systèmes de gestion du code source (SCM)

Liste partielle de termes, fichiers et utilitaires utilisés

- Dépôts de code source
- Commits, branches et tags
- Branches de fonctionnalités, de développement et de publication
- Sous-dépôts
- Fusions de code



Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	056 Collaboration et communication
Objectif :	056.2 Gestion du code source
Leçon :	1 sur 1

Introduction

Quiconque a déjà édité un document texte au sein d'une équipe connaît les problèmes liés à ce type de collaboration : quelle est la version actuelle ? Où cette version est-elle enregistrée ? Est-elle en cours d'édition par quelqu'un d'autre ? Qui a apporté quels commentaires ou modifications au texte — quand et pourquoi ? Il en résulte souvent des versions divergentes du document et, dans le pire des cas, un ensemble de versions dont personne n'a la maîtrise.

Imaginez maintenant un projet logiciel comportant des centaines de fichiers, sur lequel travaillent des développeurs du monde entier qui développent de nouvelles fonctionnalités, corrigent des bogues, scindent certaines parties pour les développer séparément, etc. Un tel processus de développement n'est plus gérable sans outils adaptés.

Un logiciel spécial dédié à la *gestion du code source* (*source code management* - SCM), également appelé *système de gestion de versions* (*version control systems* - VCS) ou *système de contrôle de révisions* (*revision control systems* - RCS), apporte ici une solution. Il élimine les problèmes qui viennent d'être décrits.

Dans le monde du développement logiciel, la gestion des versions (SCM) est un pilier fondamental

qui garantit l'intégrité de votre code-base. Imaginez-la comme un gardien assidu qui suit méticuleusement chaque modification apportée à votre code source au fil du temps.

Système de gestion du code source et dépôt

Le système de gestion du code source est au cœur d'un projet logiciel. Bien que des travaux importants soient menés dans les forums de discussion et ailleurs, le système SCM représente à la fois l'historique du projet et son état actuel en tant qu'entité vivante.

Le *dépôt* de code source est une sorte d'atelier numérique pour un projet. Tout comme un atelier physique stocke tous les outils et éléments nécessaires à la fabrication d'une pièce, un dépôt de code source stocke tous les fichiers, documents et codes liés à un projet logiciel. Il fournit un environnement structuré pour organiser et gérer les ressources du projet.

Les informations sont généralement stockées sous la forme d'une arborescence de répertoires. Les systèmes SCM utilisent habituellement un modèle client-serveur, dans lequel tout utilisateur peut *tirer (pull)* des données depuis le dépôt ou *pousser (push)* des données vers celui-ci. Le système garde une trace de l'auteur et de la date de chaque modification, garantissant ainsi la transparence et la responsabilité au sein de l'équipe.

Imaginez que vous travaillez sur un projet avec plusieurs développeurs et qu'un bogue est découvert dans le code. Grâce à un SCM, vous pouvez facilement examiner les modifications entre les versions afin d'identifier précisément le changement de code responsable du bogue.

Comme le système mémorise chaque version des fichiers à mesure qu'ils changent, l'utilisateur a accès à toutes ces versions et peut revenir à des versions antérieures (au cas où des modifications incorrectes auraient été apportées). Le dépôt est donc également une sorte d'archive, donnant accès à toutes les modifications jamais apportées au projet et à l'état du projet à tout moment de son histoire.

Les systèmes SCM permettent de gagner de l'espace en conservant une trace des modifications apportées à un fichier plutôt que de stocker le fichier complet à chaque modification. Cette méthode de stockage efficace garantit l'accessibilité des versions historiques sans consommer de ressources excessives.

De nombreux outils, tels que l'intégration continue/la livraison continue (CI/CD) et les tests, s'articulent autour du système SCM. Les utilisateurs se forgent également une réputation grâce à ce système en rendant leurs contributions visibles à tous. Par conséquent, la gestion du code source ne consiste pas seulement à suivre les modifications, mais aussi à préserver l'intégrité de votre code-base et à favoriser la collaboration entre les développeurs, afin de garantir que vos projets puissent évoluer dans un environnement dynamique et en constante mutation.

Les systèmes SCM populaires incluent Git, Subversion et CVS. Comme beaucoup d'autres fonctions logicielles, le SCM est désormais souvent fourni par des fournisseurs spécialisés. En d'autres termes, il s'agit d'un logiciel en tant que service (*Software as a Service* - SaaS) ou d'un service "cloud" : les participants disposent du logiciel sur leurs systèmes locaux pour gérer leurs modifications personnelles, mais téléversent leurs modifications vers un dépôt central qui bénéficie des fonctionnalités classiques d'un cloud telles que la disponibilité 24 heures sur 24, 7 jours sur 7, les sauvegardes et l'accès sécurisé.

Des millions de développeurs utilisent désormais les services cloud, notamment GitHub. GitLab est une alternative basée sur un code open source. GitHub et GitLab permettent tous deux aux utilisateurs de travailler dans les dépôts cloud du fournisseur ou de configurer une version locale du système SCM. L'un des avantages de ces systèmes réside dans la réputation que l'on peut acquérir grâce à leur système d'"étoiles", qui permet aux utilisateurs d'évaluer le travail des autres. Les services cloud offrent des fonctionnalités supplémentaires telles que des outils de suivi des demandes de modification, des systèmes de notation, des wikis et des forums de discussion.

Les dépôts peuvent contenir à la fois des projets personnels et d'entreprise, ce qui signifie qu'ils ne sont pas nécessairement réservés à un usage professionnel. Tout développeur qui souhaite lancer un projet de développement ou travailler sur un projet open source, en le copiant et en le modifiant selon ses besoins, peut le faire. Dans tous ces cas, il est important de contrôler strictement qui peut accéder à votre dépôt.

Il est essentiel de comprendre la terminologie relative au contrôle de version et à la gestion du code source pour pouvoir utiliser ces outils. Dans les sections suivantes, nous examinons de plus près certains de ces concepts et termes.

Commits, Tags, and Branches

Un développeur crée un *commit* chaque fois qu'une modification est téléversée dans le dépôt. Les commits représentent des instantanés des modifications apportées au code-base à un moment donné. Chaque commit comprend des métadonnées telles que le nom de l'auteur, un horodatage et un message descriptif expliquant les modifications. Les commits aident les développeurs à suivre l'évolution du code-base et à comprendre l'historique des modifications spécifiques.

Prenons l'exemple d'une équipe de développeurs travaillant sur une application web. Chaque fois qu'ils apportent des modifications au code source, ils créent un commit pour documenter ces modifications. Par exemple, une personne qui ajoute une nouvelle fonctionnalité à l'application peut créer un commit avec un message tel que "Ajout de la fonctionnalité d'authentification utilisateur". Ce commit capture l'état du code-base après l'implémentation de la fonctionnalité.

Lorsqu'un développeur corrige un bogue, le message du commit fait généralement référence au

numéro du bogue dans le système de suivi des bogues du projet.

Les *balises* (*Tags*) sont des références nommées à des commits spécifiques. Elles marquent généralement des moments importants dans l'historique du projet, tels que des versions ou des étapes clés. Les balises permettent d'étiqueter et de référencer les versions importantes du code-base, ce qui facilite la gestion et la navigation dans l'historique du projet.

Les *branches* entrent en jeu lorsque les développeurs doivent travailler simultanément sur différentes tâches. Les branches sont des lignes de développement indépendantes qui divergent du code-base principal. Elles permettent aux développeurs de travailler sur des fonctionnalités ou des corrections de manière isolée, sans affecter le code principal, jusqu'à ce qu'elles soient prêtes à être intégrées. Les branches aident à organiser les efforts de développement et facilitent la collaboration entre les membres de l'équipe.

Par exemple, si un développeur travaille à l'ajout d'une nouvelle fonctionnalité à l'application tandis qu'un autre corrige un bogue, ils peuvent chacun créer des branches distinctes afin d'isoler leurs modifications. Une fois leur travail terminé, ils peuvent *fusionner* leurs branches dans la code-base principal ([Branches](#)).

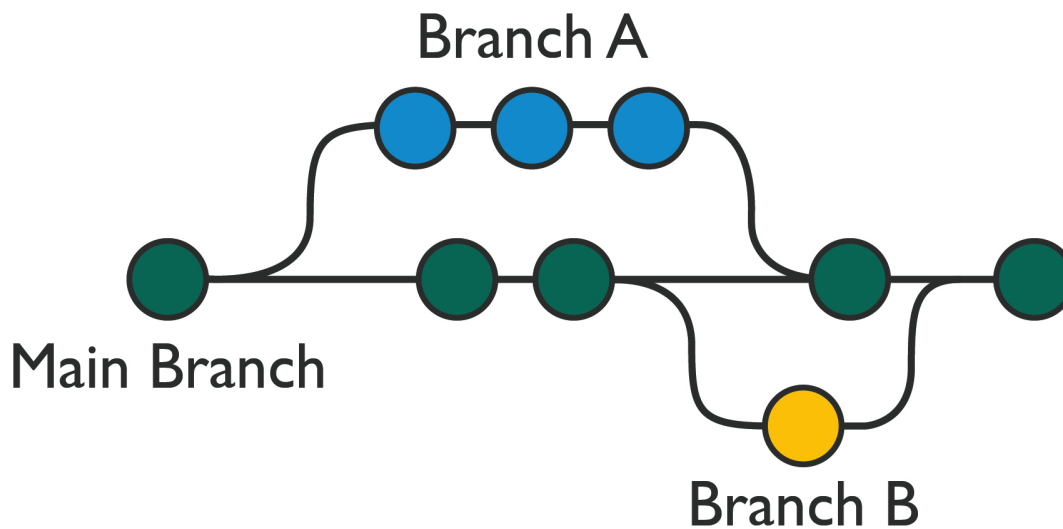


Figure 16. Branches

Lorsque plusieurs personnes travaillent sur le même fichier ou enregistrent des fichiers dans une autre branche, il arrive parfois que deux personnes modifient la même ligne d'un fichier. Lorsque la deuxième personne à avoir apporté une modification tente de l'enregistrer, le système signale un *conflit*.

Certains systèmes de gestion de versions ne permettent tout simplement pas à un développeur d'enregistrer un fichier s'il contient un conflit avec la version actuelle dans le dépôt. Le

développeur doit extraire la version actuelle et trouver comment résoudre le conflit, puis enregistrer une nouvelle version.

Les systèmes basés sur le cloud proposent des *demandes de fusion* (*merge requests*) ou des *demandes d'extraction* (*pull requests*). Celles-ci sont créées par un développeur qui a travaillé sur un système local ou une branche et estime que ses modifications peuvent être intégrées à une autre branche. Les développeurs de la branche cible décident d'accepter ou non la demande.

Sous-dépôts

Il arrive souvent que le code d'un autre projet indépendant (par exemple, un lecteur multimédia) soit nécessaire pour le développement d'un projet logiciel (par exemple, un site web sophistiqué). Au lieu de copier tout ou une partie du code du lecteur multimédia dans votre propre projet, de nombreux VCS offrent la fonctionnalité des *sous-dépôts*, également appelés *sous-modules*.

Dans notre exemple, le dépôt du lecteur multimédia est intégré au dépôt du site web en tant que sous-dépôt. Il apparaît alors comme un répertoire distinct dans l'arborescence des répertoires. Cela signifie que le code source du lecteur multimédia est entièrement disponible, mais reste indépendant. Si nécessaire, il peut même être mis à jour à partir du dépôt d'origine du lecteur multimédia.

Cette fonctionnalité s'avère précieuse pour gérer des projets complexes comportant de nombreuses dépendances ou pour intégrer des bibliothèques et des frameworks tiers dans un code-base. Les sous-modules améliorent l'organisation des projets et facilitent la collaboration en permettant aux développeurs de travailler efficacement avec des code-bases interconnectés.

Utilisation générale d'un système de gestion du code source

Chaque participant à un projet commence par créer une identité, généralement liée à son adresse e-mail unique. Un système basé sur le cloud gère les identités via des comptes, comme le font les réseaux sociaux et d'autres organisations.

Les nouveaux développeurs suivent une séquence similaire à celle ci-dessous lorsqu'ils utilisent un système SCM :

1. Installer le logiciel SCM, s'il n'est pas déjà fourni sur leur système d'exploitation.
2. Télécharger l'intégralité du projet sur le système local une seule fois, également appelé *clonage*.
3. À partir de cette étape, travailler localement (dans une ou plusieurs branches, si nécessaire) et pousser les modifications vers le dépôt.
4. Tirer la version récente du dépôt avant la prochaine session de travail.

Les chefs de projet décident à qui faire confiance et donner accès au dépôt. Les développeurs seniors ont la responsabilité importante de décider quand les modifications soumises par d'autres contributeurs sont prêtes à être intégrées dans la branche principale du dépôt.

Sur les systèmes basés sur le cloud, le propriétaire d'un projet peut contrôler l'accessibilité d'un dépôt en définissant sa visibilité comme publique ou privée. Les dépôts publics accordent un accès en lecture à toute personne connectée à Internet. Les dépôts privés, en revanche, limitent l'accès au seul propriétaire, aux personnes avec lesquelles le propriétaire a explicitement partagé l'accès et, dans le cas des dépôts d'organisation, à certains membres de l'organisation.

Imaginez une équipe de développeurs travaillant sur un site web de commerce électronique. Ils décident d'ajouter une nouvelle fonctionnalité permettant aux clients de suivre leurs commandes. Pour mettre en œuvre cette fonctionnalité, ils créent une *branche de fonctionnalité* (*feature branch*) avec un nom tel que `order-tracking`, où ils peuvent travailler sur les modifications de code nécessaires sans affecter le code-base principal. Une fois la fonctionnalité terminée et testée, ils fusionnent cette branche dans la *branche de développement* principale pour une intégration et des tests supplémentaires.

La branche de développement principale sert de plaque tournante centrale où toutes les nouvelles fonctionnalités sont rassemblées pour être testées. Par exemple, si plusieurs développeurs travaillent simultanément sur différentes fonctionnalités, ils peuvent fusionner leurs branches de fonctionnalités dans la branche de développement afin de s'assurer que tout fonctionne correctement ensemble. Ce processus d'intégration permet d'identifier et de résoudre rapidement les conflits ou les problèmes de compatibilité.

Lorsqu'il est temps de publier une nouvelle version du site Web de commerce électronique, l'équipe crée une *branche de publication* (*release branch*), telle que `v2.0`, à partir de la branche de développement. Elle se concentre sur la stabilisation du code-base, la correction des derniers bogues et la réalisation de tests approfondis afin de garantir une publication sans heurts. Une fois la publication prête, le code de la branche de publication est déployé en production, et le cycle recommence.

Systèmes de gestion de versions courants

Parmi les systèmes de gestion de versions les plus connus, on peut citer Git, Subversion (également appelé SVN) et CVS. Tous ces systèmes sont open source.

Git est un système de gestion de versions distribué largement utilisé dans le développement logiciel et dans d'autres domaines. Lorsque Git est utilisé, chaque développeur dispose d'une copie complète du code-base sur son ordinateur.

Cette approche décentralisée permet aux développeurs de travailler hors ligne et de collaborer de manière transparente, sans dépendre d'un serveur central. Ainsi, les développeurs peuvent travailler indépendamment sur différentes fonctionnalités ou corrections et fusionner leurs modifications de manière transparente. Même si le serveur central est hors ligne, les développeurs peuvent continuer à travailler et partager leurs mises à jour entre eux.

Prenons l'exemple du développement du noyau Linux, qui reposait initialement sur un système de contrôle de version centralisé appelé BitKeeper. Lorsque BitKeeper a cessé d'être gratuit, Linus Torvalds et la communauté Linux ont développé Git comme alternative distribuée. Cette décision a permis un développement non linéaire et une gestion efficace de projets de grande envergure tels que le noyau Linux. Le succès de Git pour le noyau Linux — un projet extrêmement complexe impliquant des milliers de développeurs et d'innombrables branches — démontre la puissance et l'évolutivité de Git.

De nos jours, la plupart des développements logiciels utilisent Git. Des offres SaaS extrêmement populaires sont construites autour de lui.

Git traite les conflits en fournissant au développeur un fichier contenant les deux versions de la ligne modifiée, en indiquant clairement de quelle version du fichier proviennent les lignes. Le développeur doit décider comment résoudre le conflit et enregistrer une version cohérente du fichier.

Subversion (SVN) était probablement le système SCM le plus populaire avant l'invention de Git. Contrairement à Git, Subversion est centralisé : l'historique des versions réside sur un serveur central. Les développeurs se connectent à ce serveur pour apporter des modifications, ce qui garantit que tout le monde travaille avec la dernière version du code-base.

Supposons que vous faites partie d'une équipe travaillant sur un projet à l'aide de Subversion. Chaque fois que vous devez apporter des modifications au code-base, vous vous connectez au serveur SVN central pour obtenir une copie de travail du code. Cela vous garantit de travailler avec la version la plus récente du projet. Après avoir apporté vos modifications, vous les remettez sur le serveur, mettant ainsi à jour le dépôt central avec vos modifications. Le flux de travail centralisé permet de maintenir la cohérence et garantit que tout le monde travaille vers les mêmes objectifs.

Avant Subversion, CVS était un système de gestion de versions centralisé très populaire. Il présentait des problèmes de conception qui ont conduit à la conception de Subversion comme alternative.

Exercices guidés

1. Citez trois caractéristiques essentielles des systèmes SCM.

2. Décrivez le concept de balisage dans les systèmes SCM et expliquez pourquoi il est important pour la gestion de la publication des logiciels.

3. Quelle est la différence entre une branche et un sous-dépôt dans un système SCM ?

Exercices d'exploration

1. Comparez Git et Subversion (SVN) en termes d'architecture et de flux de travail.

2. Qu'est-ce que l'"index" ou la "staging area" dans Git ?

3. Présentez la stratégie de ramification basée sur le tronc dans Git.

4. Parmi les systèmes SCM suivants, lesquels sont open source ?

Git	
Mercurial	
Subversion	
GitHub	
Bitbucket	
GitLab	

Résumé

Cette leçon a expliqué le rôle central des systèmes de gestion du code source dans le développement logiciel moderne. Vous avez appris les termes de base et les modes d'utilisation du système, notamment les dépôts, les branches, les balises et les fusions.

Réponses aux exercices guidés

1. Citez trois caractéristiques essentielles des systèmes SCM.
 - Journalisation des modifications apportées au code source
 - Gestion des accès (simultanés) au code source par les développeurs
 - Possibilité de restaurer n'importe quel état de développement des fichiers ou de l'ensemble du projet
2. Décrivez le concept de balisage dans les systèmes SCM et expliquez pourquoi il est important pour la gestion de la publication des logiciels.

Le balisage consiste à attribuer des étiquettes ou des noms descriptifs à des commits spécifiques au sein du code-base, qui servent de repères pour les moments importants de l'historique du projet, tels que les publications ou les étapes clés. Ces balises offrent un moyen pratique de faire référence à des versions particulières du code-base et de suivre l'évolution du projet au fil du temps.

3. Quelle est la différence entre une branche et un sous-dépôt dans un système SCM ?

Une branche est une ligne de développement parallèle dans un projet, par exemple pour la correction de bogues ou le développement de nouvelles fonctionnalités, qui est généralement fusionnée dans la branche de développement principale dès que la tâche est terminée. Un sous-dépôt ou sous-module est un projet indépendant dont le dépôt est intégré à un projet afin d'accéder à son code-base. Le sous-dépôt apparaît comme un répertoire dans l'arborescence du projet et reste indépendant.

Réponses aux exercices d'exploration

1. Comparez Git et Subversion (SVN) en termes d'architecture et de flux de travail.

Git est un système de contrôle de version distribué (*distributed version control system* - DVCS) qui permet à chaque développeur de disposer d'une copie complète du code-base et de travailler hors ligne sur le code source. Git a acquis une grande popularité auprès des développeurs en raison de sa rapidité, de sa flexibilité et de ses solides capacités de ramification et de fusion. SVN est un système de contrôle de version centralisé, dont l'historique des versions réside sur un serveur central. Il reste populaire dans certains environnements d'entreprise en raison de sa nature centralisée et de ses fonctionnalités matures.

2. Qu'est-ce que l'“index” ou “staging area” dans Git ?

L'index ou *staging area* est une couche intermédiaire entre la copie de travail locale du projet et la version actuelle sur le serveur. Il s'agit d'un fichier dans lequel sont stockées toutes les informations relatives au prochain commit de l'utilisateur.

3. Présentez la stratégie de ramification basée sur le tronc dans Git.

Le développement basé sur le tronc est une stratégie qui met l'accent sur l'intégration fréquente des modifications dans le code-base principal (tronc). Les développeurs travaillent sur des branches de fonctionnalités à courte durée de vie et les fusionnent dans le tronc plusieurs fois par jour, garantissant ainsi une intégration continue et un retour rapide.

4. Parmi les systèmes SCM suivants, lesquels sont open source ?

Git	X
Mercurial	X
Subversion	X
GitHub	
Bitbucket	
GitLab	X



056.3 Outils de communication et de collaboration

Référence aux objectifs de LPI

Open Source Essentials version 1.0, Exam 050, Objective 056.3

Valeur

2

Domaines de connaissance les plus importants

- Comprendre les principaux outils de communication
- Comprendre les principaux moyens d'acquisition et de sécurisation des connaissances
- Comprendre les principaux outils de gestion et de publication de l'information
- Comprendre les principaux types de documentation
- Comprendre les principales fonctions de collaboration dans les plateformes de gestion du code source
- Comprendre les concepts d'applications et de plateformes gérées de manière autonome, fédérée et centralisée

Liste partielle de termes, fichiers et utilitaires utilisés

- Messageries instantanées
- Plateformes de discussion
- Listes de diffusion
- Bulletins d'information
- Suivi des problèmes et des bogues
- Rapports de bogues
- Demandes de fusion et demandes d'extraction

- Services d'assistance et systèmes de gestion des tickets
- Wikis
- Systèmes de gestion de documents
- Sites web de documentation
- Sites web de produits
- Systèmes de gestion de contenu
- Documentation d'architecture
- Documentation utilisateur
- Documentation administrateur
- Documentation développeur



Leçon 1

Certification :	Open Source Essentials
Version :	1.0
Thème :	056 Collaboration et communication
Objectif :	056.3 Outils de communication et de collaboration
Leçon :	1 sur 1

Introduction

De nombreux projets open source comptent des participants actifs dans le monde entier. La collaboration se déroule principalement dans un “monde virtuel” et les personnes impliquées sont souvent réparties dans différents pays, continents et fuseaux horaires. De plus, elles parlent généralement des langues maternelles différentes. Cela signifie que vous pouvez vous trouver n’importe où dans le monde pour contribuer à un projet open source, quel que soit votre pays ou votre langue !

Si cette diversité rend la contribution à un projet open source extrêmement enrichissante, car elle permet d’élargir ses horizons et d’apprendre beaucoup, elle peut également rendre la communication et la coordination assez difficiles. Une communication efficace et efficiente est essentielle à la réussite et à la pérennité de tout projet open source. Pour garantir une bonne communication, les projets open source ont mis en place des structures et utilisent divers outils qui facilitent les contributions et rendent la coopération plus efficace.

Un autre défi réside dans le fait que les projets open source, souvent menés principalement par des bénévoles, sont confrontés à une certaine fluctuation dans la participation. La vie et les loisirs

des gens changent, et certains peuvent perdre leur intérêt ou simplement manquer de temps. Vous pouvez contribuer à un projet open source pendant des années, mais lorsque vous changez d'emploi, vous mariez ou commencez à élever des enfants, vous n'avez peut-être plus assez de temps pour faire du bénévolat.

Par conséquent, outre la mise à disposition de moyens de communication efficaces via des outils adaptés, les projets open source doivent garantir la préservation et le partage des connaissances afin d'éviter de "réinventer la roue". La préservation des informations aide également les contributeurs à tirer les leçons des erreurs passées commises par d'autres contributeurs et à éviter de reproduire les mêmes erreurs.

Cette leçon présente les outils courants utilisés pour coopérer dans le cadre d'un projet open source et vous initie aux moyens de communication au sein d'une communauté internationale. Vous découvrirez qu'il existe un moyen simple pour chacun d'apporter sa première contribution !

À titre d'exemple de communication et de partage d'informations, nous nous intéresserons à *LibreOffice*. LibreOffice est une suite bureautique open source disponible dans plus d'une centaine de langues. Ses utilisateurs finaux vont des particuliers occasionnels aux grands gouvernements et entreprises. De même, les contributeurs sont très variés — il ne s'agit pas seulement de développeurs, mais aussi de traducteurs, d'auteurs de documentation, de spécialistes du marketing, d'ingénieurs en assurance qualité, d'administrateurs d'infrastructure, de graphistes et de concepteurs UX, et bien d'autres encore.

En d'autres termes : pour LibreOffice, comme pour beaucoup d'autres projets open source, vous pouvez mettre à profit vos compétences et votre talent dans le domaine qui vous convient le mieux. Vous n'avez pas nécessairement besoin d'être une personne ayant une formation technique ou un développeur — vous pouvez également apporter votre créativité, votre talent artistique ou vos compétences linguistiques. Le projet a également lancé un site web qui présente les différents domaines de contribution : <https://whatcanidoforlibreoffice.org>. LibreOffice illustre donc parfaitement de nombreux aspects du travail communautaire.

Modes de communication

Avant d'examiner en détail les outils de communication, il est important de comprendre comment fonctionne la communication en général, car ces considérations influencent le choix des outils.

Les projets open source bénéficient de deux modes de communication principaux. Avec la communication *synchrone*, les personnes communiquent en même temps. Les conversations directes en sont bien sûr un exemple, mais aussi les vidéoconférences ou les appels téléphoniques. Avec la communication *asynchrone*, les personnes communiquent à des moments différents. Les e-mails et les SMS en sont des exemples, mais aussi les lettres postales ou les télécopies.

Cette distinction n'est toutefois pas toujours facile à établir. À titre d'exemple, une application de messagerie instantanée sur téléphone mobile telle que WhatsApp, Telegram, Signal ou Element est techniquement un moyen de communication asynchrone. Cependant, si les deux interlocuteurs sont en ligne au même moment et répondent instantanément, ils entrent dans une conversation directe.

Cet exemple montre qu'une partie de la communication dépend également de la manière dont les gens utilisent leurs outils et de leurs attentes. Chaque tâche peut nécessiter un ensemble différent d'outils de communication. Les sections suivantes expliquent ces idées en détail.

Communication synchrone

La communication synchrone est un mode de communication très efficace, mais aussi très exigeant. Elle rassemble les gens au même moment dans un même espace physique ou virtuel.

Une rencontre directe est idéale pour discuter de manière interactive plutôt que d'envoyer de longs e-mails qui comportent un risque de malentendus. Imaginez que vous souhaitiez en savoir plus sur un projet open source et faire connaissance avec les membres de la communauté. Les e-mails et les sites web peuvent vous aider à vous présenter et sont plus faciles d'accès, mais c'est en discutant en personne avec quelqu'un que vous ferez une première impression vraiment significative — ce sont ces interactions directes qui fascinent généralement les gens dans un projet open source et leur donnent envie d'y contribuer.

En plus de permettre de faire connaissance, les réunions synchrones sont également idéales pour discuter de manière interactive, par exemple en cas de conflits ou de problèmes, ou lorsque vous devez partager une nouvelle négative.

L'inconvénient est que les projets internationaux sont confrontés à un défi que la technologie ne peut surmonter : les fuseaux horaires. Si vos contributeurs se trouvent sur différents continents, il sera très difficile de trouver des créneaux horaires qui conviennent à tout le monde. Une personne en Australie peut être en train de se réveiller alors qu'une autre en Europe est sur le point de terminer sa journée. De plus, certaines personnes ne peuvent travailler que la nuit ou le week-end, tandis que d'autres préfèrent les heures de bureau pendant les jours ouvrables.

De plus, l'anglais n'est pas une langue que tout le monde parle couramment, et il n'existe pas encore d'outils de traduction en direct largement accessibles, ce qui constitue un obstacle supplémentaire.

Malgré tout, les vidéoconférences sont l'un des outils de communication les plus fréquemment utilisés dans les projets open source. Le projet LibreOffice, qui sert d'exemple pour cette leçon, organise régulièrement des réunions en ligne pour sa communauté, par exemple pour les développeurs, le marketing, l'infrastructure, l'assurance qualité, l'expérience utilisateur et la

conception.

Communication asynchrone

La communication asynchrone vous permet de participer à la discussion au moment et à la vitesse qui vous conviennent. L'exemple le plus connu est sans doute celui d'un e-mail : vous pouvez répondre à un message quand vous le souhaitez, que ce soit dans la minute qui suit, le lendemain ou dans l'heure qui suit.

Dans le cas de la communication asynchrone, le contenu est généralement écrit, ce qui ouvre en outre la possibilité d'une traduction automatique. Cela vous aide à lire et à comprendre les messages rédigés dans une langue différente de votre langue maternelle, et vous pouvez même traduire votre réponse.

De plus, le contenu déjà rédigé facilite grandement la conservation des connaissances et la production de documentation. Imaginez que vous souhaitiez rédiger un document d'assistance sur l'utilisation d'une fonctionnalité spécifique d'un logiciel. Si vous l'expliquez à un utilisateur par téléphone, il sera beaucoup plus difficile de transformer votre discours en une page de documentation appropriée que si vous expliquez la procédure par écrit.

Communication interne versus communication externe

Enfin, la communication dépend également des destinataires visés, c'est-à-dire qu'elle peut être interne ou externe. Vous rédigerez une note technique interne à l'intention des administrateurs système différemment d'un communiqué de presse destiné à des centaines de journalistes. Gardez toutefois à l'esprit qu'en raison de la nature d'un projet open source, les communications qui ne sont pas explicitement destinées au grand public seront néanmoins accessibles à tous, par exemple dans les archives des listes de diffusion (dans le cas de LibreOffice, il s'agit de <https://listarchives.documentfoundation.org/>).

Outils de communication

En gardant à l'esprit ces différents aspects généraux de la communication, vous découvrirez dans les sections suivantes plusieurs outils de communication couramment utilisés dans un projet open source.

E-mails, listes de diffusion et bulletins d'information

L'un des premiers outils que vous rencontrerez lorsque vous participerez à un projet open source est le courrier électronique "classique". De nombreux projets utilisent des *listes de diffusion*, qui sont essentiellement des listes de distribution pour les courriers électroniques : avec un seul

message électronique, vous pouvez atteindre des centaines, voire des milliers d'abonnés intéressés par des sujets particuliers.

Les listes de diffusion comptent parmi les outils les plus anciens connus dans tout projet open source. Elles sont utilisées pour la coordination interne au projet ainsi que pour interagir avec les utilisateurs. Si vous avez une question sur le logiciel ou si vous souhaitez signaler une erreur dans le programme, il y a de fortes chances qu'il existe une liste de diffusion qui vous permette de le faire. La communauté LibreOffice, par exemple, propose diverses listes de diffusion internationales et locales sur différents sujets (<https://www.libreoffice.org/get-help/mailling-lists/>), allant de l'assistance aux utilisateurs aux discussions entre développeurs et à la coordination des infrastructures ([Page web des listes de diffusion LibreOffice](#)).

English | 中文 (简体) | Deutsch | Español | Français

LibreOffice
The Document Foundation

DISCOVER - DOWNLOAD - GET HELP - IMPROVE IT - EVENTS - ABOUT US - DONATE Search

Get Help / Mailing Lists

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. Page web des listes de diffusion LibreOffice

Chaque message envoyé est généralement stocké dans une *archive de liste de diffusion* publique pour référence future. Une fois envoyé, un message ne peut pas être facilement supprimé. Comme le dit l'adage, "Internet n'oublie jamais". Il est donc conseillé de faire attention à ce que vous écrivez, car vous ne pourrez probablement pas le retirer. Par exemple, certaines personnes indiquent leur adresse privée ou leur numéro de téléphone dans leur signature, ou envoient des

documents confidentiels en pièces jointes, ce que vous devez éviter à tout prix.

L'un des inconvénients des listes de diffusion est qu'il n'est pas toujours facile de les gérer dans votre programme de messagerie. Vous devrez créer des *filtres* basés sur des éléments spécifiques du message, par exemple un préfixe dans la ligne d'objet. Les subtilités de la gestion d'un grand nombre d'e-mails peuvent constituer un obstacle pour les utilisateurs inexpérimentés — en particulier si votre message est ponctuel. C'est pourquoi de plus en plus de projets migrent vers des forums de discussion, que vous découvrirez bientôt.

Une forme particulière de liste de diffusion est le *bulletin d'information (newsletter)*. Si vous souhaitez rester informé des derniers développements du projet et être averti des nouvelles versions du logiciel, vous pouvez vous abonner à la newsletter et recevoir un message électronique lorsque quelque chose d'important se produit.

Forums de discussion

Outre la charge que représente la gestion des listes de diffusion dans votre client de messagerie, un autre inconvénient du courrier électronique est que de plus en plus de personnes, en particulier la jeune génération, ne sont plus très intéressées par la communication par courrier électronique. Pour cette raison, entre autres, de plus en plus de projets open source transfèrent leurs communications vers des *forums de discussion*. Le principe général est très similaire à celui du courrier électronique : chaque forum comporte différentes catégories avec des sujets spécifiques à débattre, appelés *fils de discussion (threads)*. Comme pour le courrier électronique, un forum vous permet d'entrer en contact avec le projet open source et de coordonner des activités, de faire des suggestions sur l'orientation du projet et de signaler des erreurs en tant qu'utilisateur final.

Tout ce qui est publié sur un forum est généralement visible par le grand public, tout comme dans une liste de diffusion ; mais contrairement à celle-ci, selon la configuration du forum, le contenu peut également être modifié ou supprimé ultérieurement. La facilité d'utilisation des forums est souvent supérieure à celle des listes de diffusion, en particulier pour les utilisateurs inexpérimentés. Le projet LibreOffice a commencé à convertir plusieurs de ses listes de diffusion en forums (<https://community.documentfoundation.org/>) et a depuis constaté une augmentation de la participation aux discussions.

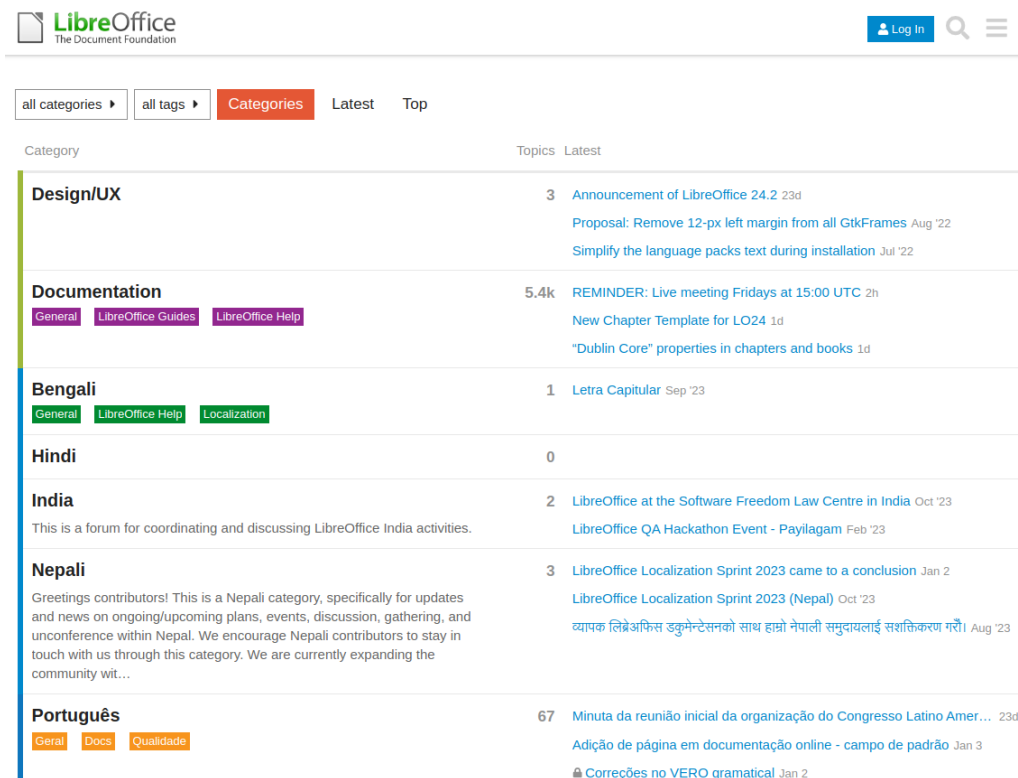


Figure 18. Page web des forums LibreOffice

Messages instantanés et plateformes de discussion

Les messageries instantanées et les plateformes de discussion constituent un autre moyen d'entrer en contact avec une communauté open source. Avec l'essor d'outils populaires tels que WhatsApp, Telegram, Signal et Matrix, presque tout le monde a déjà installé l'une de ces applications sur son appareil, ce qui réduit considérablement les obstacles à l'accès. Les messageries instantanées sont également beaucoup plus populaires auprès de la jeune génération que les e-mails ou les forums. Il n'est donc pas surprenant que de nombreux projets open source les adoptent aujourd'hui.

Les participants aux discussions tapent des messages qui sont envoyés à tous les autres participants, comme dans un e-mail. Selon la plateforme de discussion, un message peut être enrichi avec une mise en forme, des graphiques et des pièces jointes.

Structurellement, les applications de messagerie sont organisées de manière similaire à un forum ou à un service de messagerie électronique. Plusieurs *groupes* ou *canaux* sont disponibles, ce qui vous permet de participer à des discussions sur des sujets qui vous intéressent. Les messages peuvent généralement être modifiés ou supprimés, et il existe souvent des canaux réservés aux annonces qui ont une fonction similaire à celle des newsletters électroniques.

L'un des inconvénients des applications de messagerie instantanée est que les utilisateurs les

installent généralement sur leur téléphone, ce qui leur permet de recevoir des notifications pour chaque message envoyé. Cela peut rapidement entraîner un excès d'informations ou une "fatigue des alertes". Cependant, avec une configuration appropriée, ces notifications peuvent être contrôlées.

Un autre inconvénient réside dans le fait que de nombreuses applications de messagerie instantanée sont propriétaires et détenues par un seul fournisseur. Cela complique la conservation à long terme des connaissances, si les données ne sont pas librement accessibles.

Communications autonomes, fédérées et centralisées

Les projets open source fonctionnent de manière ouverte, sur la base de normes et d'outils ouverts. Il est donc essentiel de comprendre comment les différents outils sont conçus en termes d'interopérabilité. Les options peuvent être réparties en trois grandes catégories.

Une *plateforme autonome* fonctionne de manière isolée pour une seule communauté. Les forums ou les wikis, qui ne sont généralement pas connectés aux instances d'autres projets, en sont des exemples.

Les systèmes *décentralisés* ou *fédérés* fonctionnent individuellement pour chaque communauté, mais peuvent se connecter les uns aux autres. Le courrier électronique en est un exemple, car un serveur de messagerie local peut envoyer des e-mails à n'importe quel autre serveur de messagerie dans le monde. D'autres exemples incluent Nextcloud et ownCloud, qui peuvent "fédérer" des partages de fichiers avec d'autres serveurs, et le service de messagerie Element, avec lequel vous pouvez communiquer avec les utilisateurs d'autres serveurs. Les mêmes principes s'appliquent au réseau social Mastodon.

Les plateformes autonomes et distribuées présentent toutes deux un avantage considérable : le projet open source conserve le contrôle total sur l'ensemble du contenu et des fonctionnalités. Toutes les connaissances stockées dans un tel système restent la propriété de la communauté open source et ne sont pas soumises à des tiers.

Un système *centralisé*, en revanche, est géré par un seul fournisseur et n'interagit pas avec des tiers. Les exemples classiques sont les réseaux sociaux comme Facebook ou Instagram, ou les applications de messagerie comme WhatsApp ou Telegram. Tout le contenu est stocké sur les serveurs du fournisseur externe et est soumis à ses conditions générales.

Si vous êtes actif au sein d'une communauté open source, vous utilisez probablement ces trois options pour communiquer. Les systèmes centralisés sont parfaits pour entrer en contact avec d'autres personnes, car ils sont souvent populaires et comptent un grand nombre d'utilisateurs. Cependant, pour le travail concret au sein du projet, un système fédéré ou autonome contrôlé par la communauté est préférable.

Outils de collaboration

La distinction entre les outils de communication et les outils de collaboration n'est pas toujours évidente. Dans le cadre de cette leçon, l'objectif principal des outils de communication est de permettre une communication générale entre divers participants, tandis que l'objectif principal des outils de collaboration est d'aider les gens à travailler ensemble.

Les outils de collaboration adaptés permettent le stockage de fichiers, la collaboration en temps réel sur des documents, le suivi des versions et des modifications apportées aux documents entre les différentes versions logicielles, et bien plus encore. Si les e-mails ou les forums peuvent servir de stockage généraliste, les outils spécialisés rendent les connaissances plus facilement accessibles et la collaboration beaucoup plus efficace.

En d'autres termes, il s'agit d'outils spécialisés destinés à des tâches spécifiques. Si vous souhaitez contribuer à un projet open source, vous les rencontrerez très rapidement.

Wikis

L'un des outils de collaboration les plus anciens et les plus populaires est le *wiki*. Rendu célèbre notamment par Wikipédia, un wiki permet aux utilisateurs de travailler ensemble sur un site web composé de plusieurs documents ou "articles". Ceux-ci peuvent être regroupés en différentes catégories, filtrés par langue et contenir des mises en forme, des feuilles de calcul et des images.

Les wikis sont souvent utilisés comme base de connaissances à laquelle tout le monde peut contribuer. Si vous souhaitez contribuer au contenu d'un projet open source, participer à son wiki est l'un des moyens les plus simples. Vous pouvez traduire le contenu existant dans votre langue maternelle, modifier et mettre à jour des articles existants ou créer du nouveau contenu. Dans le wiki du projet LibreOffice (<https://wiki.documentfoundation.org>), vous trouverez du matériel marketing, des comptes rendus de réunions du conseil d'administration, des instructions d'installation et des informations sur la planification de conférences, le tout dans plusieurs langues ([Wiki LibreOffice](#)).

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
atalà
dansk
español
francès
galego
italiano
kurdî
latviešu
lietuvių
magyar
norsk
norsk bokmål
occitan
o'zbekcha/Ўзбекча
polski
português
português do Brasil
română
shqip
slovenčina
slovensčina
suomi
svenska
islandska
čeština
Ελληνικά
беларуская
български
монгол
русский
саха тыла
татарча/tatarça
українська
қазақша
עברית
ארום
العربية
سرائیکی
فارسی
नेपाली
हिन्दी
ਪੰਜਾਬੀ
தமிழ்
தெலுಗು
ភាសាខ្មែរ
ไทย
ភាសាខ្មែរ
ភាសាខ្មែរ
ភាសាខ្មែរ
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. Wiki LibreOffice

De nombreux projets open source hébergent également leur documentation et le système d'aide intégré à leur programme dans un wiki. De plus, les anciennes versions d'une page — appelées *révisions* –, sont archivées pour référence future.

Bien que les wikis puissent également stocker des fichiers de projet, il existe de meilleurs outils à cet effet, comme vous le découvrirez dans cette leçon.

Suivi des bogues et des problèmes

Les outils fréquemment utilisés dans les projets open source sont les outils de *suivi des bogues* (*bug trackers*), également appelés *suivi des problèmes* (*issue trackers*). Si vous découvrez un problème dans le logiciel ou souhaitez suggérer une nouvelle fonctionnalité, vous pourriez être tenté d'envoyer simplement un e-mail à ce sujet. Cependant, grâce à un outil dédié tel qu'un outil de suivi des bogues, vous pouvez fournir toutes les informations nécessaires et les étapes permettant de reproduire un problème de manière structurée, ce qui facilite la tâche des développeurs pour reproduire le problème. Un tel rapport structuré est appelé *rapport de bogue*.

De plus, les informations ne sont pas perdues et le projet n'oublie pas le problème ; il peut l'attribuer à la bonne personne et voir combien de bogues sont en cours de traitement ou ont été corrigés.

Le projet LibreOffice met à disposition un outil de suivi des bogues ouvert à tous (<https://bugs.documentfoundation.org>).

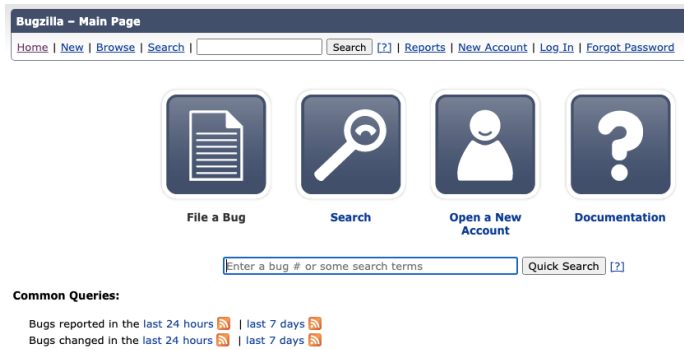


Figure 20. Suivi des bogues LibreOffice

Services d'assistance et systèmes de gestion des tickets

Un outil très similaire est le *services d'assistance (helpdesk)* ou *système de gestion des tickets*. Il est moins axé sur le signalement des problèmes logiciels que sur l'aide aux utilisateurs pour toutes sortes de questions et de demandes, par exemple concernant le site web du projet.

Le service d'assistance classique est une hotline de service à la clientèle. Un client appelle et signale un problème, qui est transformé en *ticket*. Le flux de travail d'un service d'assistance s'articule généralement autour des priorités, des niveaux hiérarchiques et des délais de réponse.

Toutes les communautés open source ne proposent pas un tel système, mais de nombreuses entreprises commerciales le font. Pour le projet LibreOffice, par exemple, il existe un système de tickets pour l'infrastructure permettant de signaler les problèmes liés aux serveurs et aux services web.

Système de gestion de contenu

Un autre outil important pour la collaboration est le *système de gestion de contenu (content management system - CMS)*. Comme son nom l'indique, il aide à gérer le contenu, notamment celui des sites web. Si vous souhaitez contribuer au contenu ou à la conception du site web d'un projet, il est préférable de vous familiariser avec son CMS.

Les systèmes CMS, similaires aux wikis, permettent de structurer le contenu par catégories et par langues. Ils fournissent souvent un éditeur WYSIWYG (“what you see is what you get” pour “ce que vous voyez est ce que vous obtenez”) et intègrent tout dans un modèle approprié : les titres, les en-têtes, la mise en page et les entrées de menu sont créés automatiquement, ce qui vous permet de vous concentrer pleinement sur le contenu. De plus, en cas de refonte d'une page, le contenu ne disparaîtra pas, mais sera adapté au nouveau modèle.

Système de gestion des documents

Il ne faut pas confondre un *système de gestion des documents* (*document management system* - DMS) avec un système de gestion de contenu (CMS). Alors qu'un CMS est conçu pour afficher du contenu dans un modèle prédéfini, comme pour présenter un site web, un DMS est utilisé pour la gestion de documents tels que les contrats, les factures, les reçus, les e-mails et tout autre type de correspondance.

Une autre différence réside dans le fait qu'un CMS est souvent utilisé pour la présentation publique, tandis qu'un DMS sert principalement à gérer des documents internes qui ne sont pas destinés au grand public.

En tant que contributeur à un projet open source, vous êtes moins susceptible d'être confronté au système de gestion des documents, car celui-ci est souvent réservé à des rôles spécifiques tels que la comptabilité ou le service juridique.

Gestion du code source

Si vous êtes développeur dans un projet open source, l'un des outils clés que vous utiliserez rapidement est la plateforme de *gestion du code source* (*source code management* - SCM). Comparables à un wiki pour les auteurs de documentation, les SCM sont utilisés par les développeurs de logiciels pour travailler ensemble sur le code.

Les SCM historiques comprennent CVS et Subversion, mais aujourd'hui, Git est le plus utilisé, y compris par la communauté LibreOffice (<https://git.libreoffice.org/>). Ces outils sont disponibles en ligne de commande, mais il existe également des interfaces graphiques qui facilitent l'interaction, en particulier pour les débutants.

Les plateformes de gestion du code source suivent les différentes versions de chaque fichier, gèrent les modifications et les corrections apportées au code, enregistrent qui a effectué quelle modification et, dans l'idéal, fournissent un historique complet du développement du logiciel.

Les développeurs peuvent "extraire" un état spécifique du logiciel, travailler localement dessus, par exemple en corrigeant un bug ou en implémentant une nouvelle fonctionnalité, puis demander que cette modification soit ajoutée à la ligne de développement principale du logiciel au moyen d'une *demande de fusion*, également appelée *demande d'extraction*. L'acceptation de la demande de fusion ou d'extraction "fusionne" les modifications apportées par un auteur avec le code principal.

Il existe des plateformes centralisées (GitHub et GitLab) qui intègrent la gestion du code source à des wikis, des outils de suivi des problèmes et d'autres outils collaboratifs.

Les plateformes de gestion du code source ne se limitent pas strictement au code des programmes. À titre d'exemple, cette leçon a été élaborée de manière collaborative dans un dépôt Git !

Documentation

La clé du succès de tout projet open source réside dans une documentation adéquate, idéalement disponible en plusieurs langues. Le projet LibreOffice fournit des manuels, des guides et des fiches de référence (<https://documentation.libreoffice.org>) à jour pour ses logiciels, ainsi que des pages d'aide individuelles sur des fonctions spécifiques (<https://help.libreoffice.org>).

En général, il existe différents types de documentation, que nous aborderons dans les sections suivantes.

Documentation utilisateur

La documentation destinée aux utilisateurs finaux, qui explique comment utiliser le logiciel, est la plus connue. Si vous ne connaissez pas une fonctionnalité ou une fonction du programme, la documentation—qui peut aller de pages d'aide individuelles à un livre complet—est votre premier point de référence.

Le site web de documentation, où l'utilisation du logiciel est expliquée, est souvent l'un des sites web les plus fréquentés après le site web du produit, qui présente une vue d'ensemble du logiciel et de sa communauté.

Documentation administrateur

Pour une utilisation dans des environnements plus vastes, comme les déploiements dans une entreprise, la documentation destinée aux administrateurs fournit toutes les informations pertinentes pour déployer le logiciel à plus grande échelle. Cela inclut la connexion aux bases de données utilisateur et au stockage de fichiers, la gestion centralisée de la configuration et la gestion des mises à jour.

Documentation pour développeurs et d'architecture

Une autre catégorie de documentation s'adresse aux développeurs et est appelée documentation pour développeurs ou documentation d'architecture. Si vous êtes développeur et souhaitez contribuer au code d'un programme, cette documentation vous informe sur l'architecture logicielle, les normes de codage, ainsi que les outils et les flux de travail utilisés pour travailler sur le logiciel.

Le projet LibreOffice a publié un guide destiné aux développeurs sur son wiki afin d'aider les

développeurs intéressés à rejoindre la communauté (
<https://wiki.documentfoundation.org/Documentation/DevGuide>).

Exercices guidés

1. Pourquoi les projets open source doivent-ils particulièrement veiller à disposer d'outils adaptés pour la communication et la collaboration ?

2. Donnez un exemple de communication synchrone et un exemple de communication asynchrone.

3. Quel est l'inconvénient des applications de messagerie instantanée et comment peut-on l'éviter ?

4. Citez deux fonctions d'un wiki.

5. Quelle est la différence entre un outil de suivi des bogues et un système d'assistance ?

6. Quelle est la différence entre un système de gestion de contenu et un système de gestion de documents ?

7. Quel est l'avantage des systèmes autonomes ou fédérés par rapport aux systèmes centralisés ?

Exercices d'exploration

1. Quelle est l'une des principales différences entre un club sportif local et un projet open source international ?

2. Pourquoi contribuer à un projet open source peut-il être particulièrement gratifiant ?

3. Quel logiciel de suivi des bogues le projet Ubuntu utilise-t-il ?

4. Quel est le nom du site web des listes de diffusion du noyau Linux ?

Résumé

Dans cette leçon, vous avez découvert divers outils utilisés pour communiquer et collaborer dans le cadre d'un projet open source. Vous avez appris la différence entre la communication synchrone et asynchrone, ainsi qu'entre les outils décentralisés, centralisés et autonomes. Vous avez également compris pourquoi certains outils peuvent être utiles pour des tâches spécifiques afin de rendre la contribution à un projet open source agréable et enrichissante.

Réponses aux exercices guidés

1. Pourquoi les projets open source doivent-ils particulièrement veiller à disposer d'outils adaptés pour la communication et la collaboration ?

D'une part, travailler ensemble au sein d'un groupe réparti dans le monde entier comporte son lot de défis, que des outils adaptés peuvent aider à relever. D'autre part, les bénévoles ne restent pas forcément indéfiniment, c'est pourquoi la conservation et le partage des connaissances constituent un autre aspect important de l'utilisation des outils de communication et de collaboration. Faciliter les contributions favorise la pérennité d'un projet.

2. Donnez un exemple de communication synchrone et un exemple de communication asynchrone.

La communication synchrone peut prendre la forme d'une conversation directe, d'un appel téléphonique ou d'une vidéoconférence. Parmi les exemples de communication asynchrone, on peut citer les e-mails, les SMS, les lettres postales et les fax.

3. Quel est l'inconvénient des applications de messagerie instantanée et comment peut-on l'éviter ?

Une fois installée sur votre téléphone, vous risquez de recevoir de nombreuses notifications, une pour chaque nouveau message. Une configuration adéquate peut vous aider à éviter cela. Un autre inconvénient est que de nombreuses applications de messagerie instantanée sont gérées par des fournisseurs propriétaires.

4. Citez deux fonctions d'un wiki.

Édition et traduction collaboratives d'articles.

5. Quelle est la différence entre un outil de suivi des bogues et un système d'assistance ?

Un outil de suivi des bogues est un logiciel spécialisé permettant de signaler des bogues ou de demander des fonctionnalités dans un logiciel. Un système d'assistance se concentre sur le traitement des requêtes et la gestion de tous types de problèmes et de demandes, par exemple sur le site web.

6. Quelle est la différence entre un système de gestion de contenu et un système de gestion de documents ?

Un CMS sert à présenter du contenu d'une manière spécifique ou selon un modèle donné, principalement à des fins publiques. Un DMS sert à stocker la correspondance existante,

principalement à des fins internes.

7. Quel est l'avantage des systèmes autonomes ou fédérés par rapport aux systèmes centralisés ?

Un système centralisé est sous le contrôle d'un seul fournisseur externe. Tout le contenu est stocké sur les serveurs du fournisseur externe et soumis à ses conditions générales.

Réponses aux exercices d'exploration

1. Quelle est l'une des principales différences entre un club sportif local et un projet open source international ?

Les projets open source ne sont pas liés à une langue ou à un lieu spécifique. Les contributeurs peuvent vivre dans différents pays et continents, avoir des langues maternelles différentes et même vivre dans des fuseaux horaires différents. La plupart des activités menées dans le cadre d'un projet open source se déroulent virtuellement, et non en présentiel.

2. Pourquoi contribuer à un projet open source peut-il être particulièrement gratifiant ?

De nombreux projets open source regroupent des personnes très diverses. En collaborant avec elles, vous pouvez apprendre, découvrir de nouvelles choses et élargir vos horizons.

3. Quel logiciel de suivi des bogues le projet Ubuntu utilise-t-il ?

Launchpad

4. Quel est le nom du site web des listes de diffusion du noyau Linux ?

<https://lkml.org/>

Impression

© 2025 par Linux Professional Institute: Supports de cours, “Open Source Essentials (Version 1.0)”.

PDF généré: 2025-09-25

Cette oeuvre est placée sous licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification 4.0 International (CC-BY-NC-ND 4.0). Pour consulter une copie de cette licence, visitez

<https://creativecommons.org/licenses/by-nc-nd/4.0/>

Bien que le Linux Professional Institute ait fait tout son possible pour s'assurer que les informations et les instructions contenues dans cette publication soient exactes, le Linux Professional Institute décline toute responsabilité en cas d'erreurs ou d'omissions, y compris, mais sans s'y limiter, la responsabilité des dommages résultant de l'utilisation ou de la confiance accordée à cette publication. L'utilisation des informations et des instructions contenues dans cet ouvrage se fait à vos risques et périls. Si les exemples de code ou toute autre technologie décrite ou contenue dans cet ouvrage sont soumis à des licences open source ou aux droits de propriété intellectuelle de tiers, il vous incombe de veiller à ce que leur utilisation soit conforme à ces licences et/ou ces droits.

Les supports de cours du LPI sont une initiative du Linux Professional Institute (<https://lpi.org>). Les supports de cours et leurs traductions sont disponibles à l'adresse <https://learning.lpi.org>.

Pour toute question ou commentaire sur cette édition ou sur l'ensemble du projet, contactez-nous à l'adresse suivante : learning@lpi.org.