Code Explorer's Guide

to the

Open Source Jungle

ANOOP THOMAS MATHEW

@atmb4u

Code Explorer's Guide to the Open Source Jungle

Getting started with open source, and quick start guide on developing software in collaboration.

Anoop Thomas Mathew

This book is for sale at
http://leanpub.com/opensourcebook

This version was published on 2014-05-25

Tweet This Book!

Please help Anoop Thomas Mathew by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought Code Explorer's Guide to the Open Source Jungle by @atmb4u

The suggested hashtag for this book is #openlearnability.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#openlearnability

Contents

Preface

"Oh! Please don't ask me to read a book, and that too, a book on open source!" - I do understand the feeling that's running through your head right now. Please bear with me, I'll try to be interesting, and the maximum extent be away from being boring and being philosophical.

This book is a boiled down version of my experiences with open source software for past years, as a user, tester, developer, creator and contributor. Aim is to educate the readers about what all can be done with open source, and explore the untapped possibilities existing with it.

I will trying to be generic while going through this book, without specifying names and projects, since my aim is to emphasis the processes and principles which are commonalities for most of the software engineering projects.

Chapter Description

Each chapter has been designed to be independent of other chapters, and conveys a single idea throughout the chapter. If you feel that ideas discussed in the chapter is something you already know or not relevant to you at the moment, feel free to skip ahead to the next chapter.

Contents

1 About this book

Checking each update on facebook, twitter and other social sharing applications, there was very little time to be aware of the software development process happening; or at least, the exposure has been minimal. Used to be very curious about things; how does a car work, what does a gear do, why do things work the way they do, and not otherwise. A lot of questions.

I had the same questions with the computers I'm working on now. But some how, I just couldn't find enough time to get into the zone. To learn more on how software works. How the pieces of code, gets shape and form, handles terabytes of data, data changes shape and becomes all these images, and transforms itself to videos and web pages!

I was wonder struck!

Wanted to know more about it, but the coziness by the mainstream operating systems and all the abstractions! I want to learn about the magic that happens. I'm curious.

Welcome to the world of curious people. Welcome to the world of collaboration. **Welcome to open source**.

> Open Source provides a shortcut to learning deep about technologies buried within the software we use everyday. It let's you do open heart surgery on a

> software.

I'm not here to tell you the difference between open source and Free Software. I'm not here for the philosophical preaching. I'm not here to evangelize the war of open source projects on properitory software. **I'm here to guide you in learning the internal workings of epic software and enable you to create and help creating elegant software projects, built by people across the world, which will touch billions of lives and change the course of history.**

1.0.1 About the Author

Anoop Thomas Mathew is the Director of R&D, Profoundis Inc.[1], and leads the engineering efforts on products like Vibe[2] and iTestifyIt[3]. He is a Technology Evangelist, Designer at Heart, Public Speaker, Python & JavaScript Hacker. Loves Music, Books and Drawing. He has spoken at conferences like Fifth Elephant 2012, FOSSMeet 2011, PyCon 2012, FOSSMeet 2013, DevCon 2013, PyCon 2013 to name a few. More info: infiniteloop.in[4]

[1] http://profoundis.com/
[2] https://vibeapp.co/
[3] http://itestifyit.com/
[4] http://infiniteloop.in/

2 Collaboration is Communication

As we discussed earlier, open source is all about community, and while collaborating in a community, first and foremost thing to be kept in mind is, everyone else's time is as valuable as yours. So try avoiding confusions as much as possible as it leads to waste of time, of the entire community.

2.1 Channels of Communication

In software, collaboration happen mostly online, through IRC/IM chats, mailing list emails, forums, wiki pages and documentation.

2.1.1 IRC

IRC or Internet Relay Chat is an Instant Messaging platform over which most of the popular open source community have a channel. This is where general discussions

as well as weekly meetings are conducted. Its no different from an IM, but people tend to respond a bit delayed.

There are popular clients while allows users to connect to corresponding channels on appropriate servers. One of the most common IRC server provider is Freenode[1] . They have got an browser version of IRC client.

2.1.2 Mailing List

A mailing list is a group e-mailing facility, where open source developers and users discuss over development and on bugs or usage support.

All the news regarding the new releases as well as new committers, weekly/monthly meeting minutes would be available in the mailing list logs. This is helpful in learning the history of the project for a budding developer.

2.1.3 Documentation

Primary source for any information regarding the project. Is a necessary part of any open source project. It will have detailed explanation regarding installation and usage of the software. Extension and also the developer guide for developers who intent to contribute to the project. Usually a reference section is also included with the documentation.

[1] http://freenode.net/

2.1.4 Tutorials and HOWTOs

These are blog posts or similar documents, explaining to a newbie, how to get on track with the software. They should be light on language and technology. Steps must be as simple and minimalistic as possible. Screencasts also works great as beginner tutorials.

2.1.5 Website

Homepage and front facing page for the project. Needs to express the *what, why* and *how* explicitly. All links to the *source code, bug tracker, mailing list, documentation, tutorials* etc, must be provided in the website.

2.2 You are what you write

The mostly used medium is what you write, and that makes it very clear that you should be good enough in written communication to be heard. Some well established written communication netiquette are explained below.

- Be concise. Get to the point. Many of the members on the lists are hard at work coding for one company or another. Some of those coders take breaks to check these lists to see what's going on and how they can help. They don't have time to read through your life long journey.
- Be specific. Before you post your problem make sure you have collected the relevant data. If you have

a video problem make sure you know the video card you are using, the driver you have installed, the version of the kernel you are using, and which release of the distribution you use. The more relevant information you give the more likely you will get help.

- No Flame wars - Which one is better is a very person to person question, and trying to win someone who doesn't agree with your ideaologies are very likely waste of time. Its better to explain project ideologies in a wiki, and avoid unproductive discussions.

Be Constructive

Man 1: "You know what, I use Software X. Its much faster and stable than other distributions" Man 2: "Do you have this Q feature? This Software M got the that. You must check it out" Man 1: "No ways. I'm sure Software X is the best. It has got W feature." Man 2: "M has got R, S and T features, do you have any idea?" ... (and a lot of further waste of time.)

- Do NOT top post. What is top posting? Top posting is replying to an e-mail where your reply sits neatly at the top of the reply email. Instead, either reply at the bottom of the e-mail or in line. The main reason for this is so people can follow the thread of conversation

that has taken place. Even though this may sound trivial it is taken very seriously in Linux and open source mailing lists. You will be ignored if you top post constantly.

- Do not insult people's grammar. Yes, there are grammar police all over the place. But the one thing you must remember is that many of these lists are populated by people who use English as a second language. So mocking someone who may wind up saving your skivvies some day is not the best way to make friends.
- Do not post off topic. This is another issue that resides just under top posting for most annoying to avid mailing list denizens. Keep your posts on topic. Of course there will be the occasional off topic post you will need to get out. For those instances make sure you start the subject with "OT:"
- Do not hijack threads. If a thread spawns a new topic for you, post that new topic in your own thread. Never steal someone else's thread from them.
- Emoticons do not sweep away an insult. You can not reply, "But didn't you see my ;-) indicating I was jk? OMG! WTF?" You see where this leads?

Some developers, when faced with fixing, or adding a feature to an open source project are under the mistaken impression that the first step before any fixing takes place, or before adding a new feature takes place is to make the code "easier for them" to work on.

"Easier for them" usually is a combination of renaming methods, fields, properties, locals; Refactoring of methods, classes; Gratuitous split of code in different files, or merging of code into a single file; Reorganization by alphabetical order, or functional order, or grouping functions closer to each other, or having helper methods first, or helper methods last. Changing indentation, aligning variables, or parameters or dozen other smaller changes.

This is not how you contribute to an open source project.

When you contribute fixes or new features to an open source project you should use the existing coding style, the existing coding patterns and stick by the active maintainer's choice for his code organization.

The maintainer is in for the long-haul, and has been working on this code for longer than you have. Chances are, he will keep doing this even after you have long moved into your next project.

Sending a maintainer a patch, or a pull request that consists of your "fix" mixed with a dozen renames, refactoring changes, variable renames, method renames, file splitting, layout changing code is not really a contribution, it is home work.

The maintainer now has to look at your mess of a patch and extract the actual improvement, wasting precious time that could have gone to something else. This sometimes negates the effort of your "contribution".

If you really have an urge to refactor the code, first of all, discuss the changes with the maintainer with the

rationale for the changes. If the maintainer agrees with the changes, make sure that you keep your refactoring and changes independent from code fixes, it makes reviewing the code a lot simpler.

The alternative, to keep your fork, is usually a guarantee that your effort will be wasted, and won't help other users. People have tried to do this. It is attempted every year, by hunders of developers who in the back of their minds are thinking "I can do better" and "I won't make the same mistakes".

So respect the original coding style, and if you want to make refactoring changes, discuss this with the maintainer.

3 Running the Show

Starting to play around with a community is interesting, but more than a interesting game as soon as the project gets popular. It needs careful nurturing at each stage to keep the project alive and on schedule. Some ideas on how an entire open source project lifecycle works is discussed in this stage.

3.1 Ideation

Everything starts with ideation stage. To get an idea is easy, but to validate whether it is interesting to others as well is a very difficult process. This is not a necessary thing in open source, while something being in the open source domain, is supposedly used by a lot of people and others contribute in the general development workflow discussed in the previous chapter.

Ross Gardler, former Manager of *OSS Watch*, believes that the scope of an open source project can be formalised by looking at a number of key attributes that characterise an open development community, namely:

- a deep level of user engagement: if you don't have users then there is no point having a project.

- collaboration: a means of working within a diverse group of people, something that the Internet has obviously made easier.
- agility: once work begins and there is a serious engagement with users, ideas and plans may need to change.
- sustainability: having the capacity to keep developing an application solution over the necessary period of time.
- tools: wikis, bug and version-trackers and email lists to support the development of the community and keep track of intellectual property rights, if relevant.

3.2 Packaging and Tool-Chains

Development activity on open source projects is constant and concurrent among many developers. Sometimes the changes made by one developer interfere with those made by another developer. Often the version control practices used in the open source community resolve these conflicts, but sometimes they do not. Frequent automated builds and tests of the software being developed are powerful tools that help catch logical conflicts early.

3.3 Managing Community

Software developers spend a large part of their time communicating with each other. Clear and effective technical

communications are needed to keep the team in sync and to allow individuals with key knowledge to apply that knowledge where it is needed.

One tenet of the open source community is that technical communications should take place in public forums. Mailing lists are the backbone of open source communications. Beyond that, open source projects need support for precisely communicating technical details and for group decision-making.

There can be weekly/monthly/yearly meetup (online/offline) for the developers in order to discuss the progress and also, future plans. Every task should be assigned to someone in the community, so that everyone is responsible for some part or the other.

3.4 Version Control

Every software development project needs version control. This is very important to keep track of the history as well as maintaining integrity while multiple person are working on the same code from different geographical locations at the same time. Open source projects need strong and flexible support for many concurrent developers working on overlapping sets of files.

3.5 Documentation

Document Every Line

It might sound absurd, but in a few weeks, even the code written by you will look like gibberish.

Let others know what's in your head. Open source projects also need standard templates for design documents, technical documentation, and end user documentation. A well-organized project should facilitate distributing these documents. But, the nature of open source projects demands that these files are globally accessible, and that administrative overhead be minimized and responsibility be spread out over the members of the development community. A central wiki could serve as an ideal candidate for collating all such documentation, tutorials, and guides.

3.6 Goals and Milestones

Every project needs explicit goals, resources, and a schedule. The open source community has addressed these issues in a uniquely flexible way. Shared "to do" lists keep track of tasks that need to be done. Personal "to do" lists keep developers on track. Milestone lists set flexible deadlines for individual features based on feedback from users and developers.

3.7 Testing; without, you fail.

Its like doing the first flight with a 1000 souls on board. The chances of going under is very high, and without any previous consideration, you are stepping into disaster. **Enter Testing**.

Testing is an integral part of the development, and ideally, open source projects, which is being developed from different locations all over the globe, *should have a code coverage of 100%*. Meaning, each line of code should be accompanied by a test.

Wait! what is testing by the way?

Lets consider a case here. Person A and Person B is working on the same project. Person A is working on feature 1 and person B on feature 2. Person A finished feature 1 and moves on to feature 3. In the mean time, Person B finishes feature 2, and tries to include his changes. Problem here is Person B's feature 2 is breaking Person A's feature 1.

What can we do about it? Before Person B includes his new set of changes, he runs the test suite, which contains testing cases for the features Person A has added. Now that there exists tests, its easier for Person B to identify

that feature 2 is breaking the code, he can make changes so that feature 1 is also intact.

Even if all testing practices are prominent and popular in the open source world, I'd recommend unit test or module testing as a mandatory. This ensures that each module is working perfectly. One thing to be remembered while attempting to write a test is to make sure you are focusing on the things that are likely to break, and are depended on external factors, like a global variable, or a user input. A good test would consist of all possible cases that will completely cover all the features.

Write test for the feature you add

It is ideal, recommended, and in some communities mandatory for each of the feature addition to be accompanied with comprehensive test cases.

Software engineering practices are key to any large development project. This site provides a home to conduct open source or collaborative software projects integrated with a set of hosted development tools to facilitate project teams adopting the "best practices" that have proven themselves in the open source community.

3.8 Quality Assurance

Open source software products have achieved a remarkable degree of quality. This is something that the closed source development world has found to be one of the most difficult and costly aspects of software development.

Developers are self-selected by their interest and knowledge of the application domain. Requirements are tacitly understood by developers who are themselves users of the software. Technical communications (including bug reports) are conducted in public. The public nature of open source helps developers take pride in their successes and think twice before releasing faulty code.

Furthermore, debugging is much more effective when split among many people who have diverse knowledge and accessbility. For example, one person might have a non-UNIX machine. He may not know anything about programming or development, but he can make sure that the software which he uses is working exactly as intended and report bugs, if any.

3.9 Knowledge Management

Knowledge is the valuable resource that sets experienced developers apart from novice ones. Sharing knowledge effectively has been key to the success of the open source community. Explicitly managing knowledge can help reduce the learning curve for novices which reduces the bar-

riers to entry for potential contributors while automatically keeping the load on the experts down to a minimum in terms of training others.

3.10 Stepping Down

Over time, either the core members' priorities might have changed or better and more competent contributors joins the team, the core members' should realize the fact, and step down. This is not an option; this is a necessity. A necessity to avoid making the project an *'abandonware'*.

3.11 Tips for Creating good open source Software

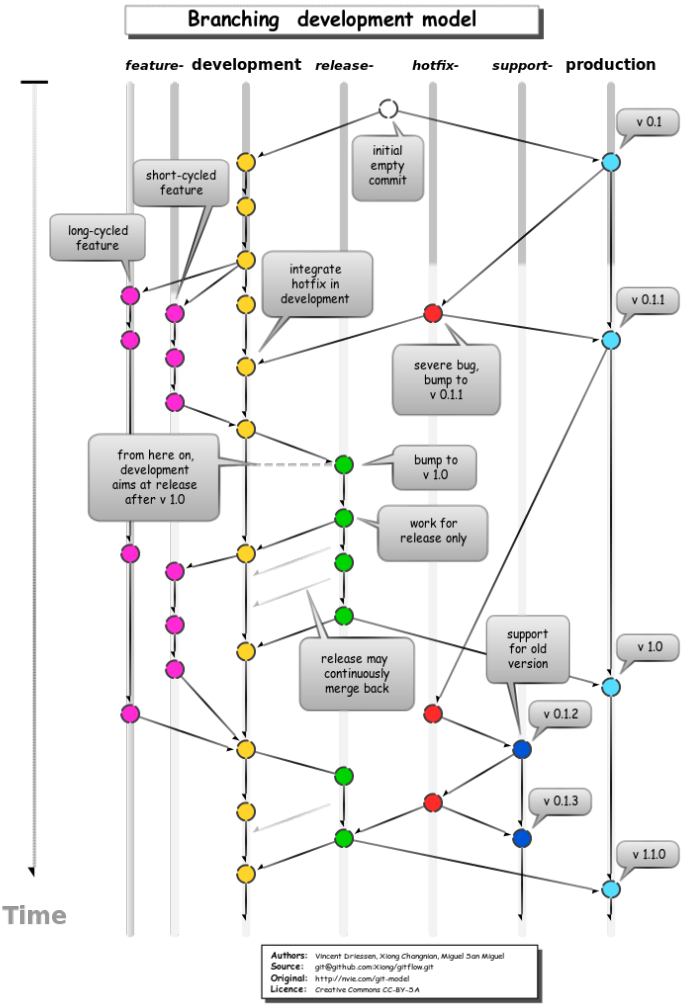
Eric S. Raymond in his book "**The Cathedral and the Bazaar**" points to 19 lessons learned from various software development efforts, each describing attributes associated with good practice in open source software development as follows:

1. Every good work of software starts by scratching a developer's personal itch.
2. Good programmers know what to write. Great ones know what to rewrite (and reuse).
3. Plan to throw one [version] away; you will, anyhow. (Copied from Frederick Brooks' The Mythical Man Month)

4. If you have the right attitude, interesting problems will find you.
5. When you lose interest in a program, your last duty to it is to hand it off to a competent successor.
6. Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging.
7. Release early. Release often. And listen to your customers.
8. Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone.
9. Smart data structures and dumb code works a lot better than the other way around.
10. If you treat your beta-testers as if they're your most valuable resource, they will respond by becoming your most valuable resource.
11. The next best thing to having good ideas is recognizing good ideas from your users. Sometimes the latter is better.
12. Often, the most striking and innovative solutions come from realizing that your concept of the problem was wrong.
13. Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away. (Attributed to Antoine de Saint-Exupéry)

14. Any tool should be useful in the expected way, but a truly great tool lends itself to uses you never expected.
15. When writing gateway software of any kind, take pains to disturb the data stream as little as possible—and never throw away information unless the recipient forces you to!
16. When your language is nowhere near Turing-complete, syntactic sugar can be your friend.
17. A security system is only as secure as its secret. Beware of pseudo-secrets.
18. To solve an interesting problem, start by finding a problem that is interesting to you.
19. Provided the development coordinator has a communications medium at least as good as the Internet, and knows how to lead without coercion, many heads are inevitably better than one.

3.12 Version Control and the git Workflow



Git Branching Workflow

As shown in the picture, a branched workflow needs to be devised with a distributed version control software, ensuring a conflict-free software development process, also maintaining the backward compatibility and strictly following agile development model.

4 Licenses FAQ

Every single line of code written by a programmer is copyrighted by default.

Yes! If any one wants to share the code with anyone else, he has to explicitly mention the license agreement. And this gave rise to all the GPL and BSD and other licenses.

4.1 Available License Types

Capabilities (Without Application Licensing Restriction)	GPL (Linux)	Dual-GPL (MySQL)	LGPL/MPL (OpenOffice, Firefox)	Apache/BSD (Apache, FreeBST)
1) Download	✔	✔	✔	✔
2) Evaluate	✔	✔	✔	✔
3) Deploy	✔	✔	✔	✔
4) Redistribute	⊘ [1]	✔ [3]	✔	✔
5) Modify	⊘ [2]	⊘ [2]	⊘ [2]	✔ [4]

1) Application needs to be licensed under GPL if redistributed with the GPL asset.
2) Library code modifications need to be licensed under the same license as the originating asset.
3) Usually requires a commercial license from the copyright holder.
4) Although much more permissive than an OSI license, some BSD based licenses, such as Apache V2, still have some copyleft materials.

Comparison of popular Software Licenses

There is an interesting perspective in haaked.com on different types of software licenses.

According to it, there are only 4 types software licenses available

1. Proprietary - The code is mine! You can't look at it. You can't reverse engineer it. Mine Mine Mine!
2. GPL - You can do whatever you want with the code, but if you distribute the code or binaries, you must make your changes open via the GPL license.

3. New BSD - Use at your own risk. Do whatever the hell you want with the code, just keep the license intact, credit me, and never sue me if the software blows your foot off. The MIT license is a notable alternative to the New BSD and is very very similar.
4. Public Domain - Do whatever you want with the code. Period. No need to mention me ever again. You can forget I ever existed.

4.2 What Kind Of license Should I Choose?

This section is based on an article by Rowan Wilson, which you can read online at *http://oss-watch.ac.uk/resources/licdiff*.

There are many free and open source software licenses, and while they all broadly attempt to facilitate the same things, they also have some differences. Some of the major differences can be grouped together into categories, and this document acts as an introduction to these categories. Having read this document, you should be able to understand which decisions you should take in order to select a license for your code.

4.2.1 Staying Mainstream

The Open Source Initiative - a non-profit organisation formed to educate about open source - maintains a list of licenses that they see as being ‘popular and widely used

or with strong communities'. The purpose of the list is to highlight those licenses which are likely to answer the needs of many licenses new to open source. The list also helps to make the task of open source license selection seem manageable; the full list of more than 60 licenses can be daunting, and contains many licenses which are in most ways similar in function to one or more of the 'popular and widely used' listed licenses.

Using a license that many others use has some advantages. If the terms of the license are challenged, there will be a larger pool of licensees with an interest in funding a response to the challenge. Using one of the 'popular and widely used' licenses also makes it more likely that your licensees will already be familiar with the terms you are offering. Equally though, licensees should not feel 'locked in' to only using a 'popular and widely used' license. Some of the features we discuss below are only present in the wider pool of all OSI-approved licenses.

4.2.2 Permissive And Copyleft

All free and open source licenses allow others to make modified versions of your code, and to make these modified versions available to others. The license you choose can make conditions about how this happens - specifically what licenses can be used on these modified versions. These conditions can help keep your code free, but they can also put some people off reusing your code. Such conditions are sometimes called 'copyleft' conditions, in a play on the

word ‘copyright’.

Open source licenses that do not seek to control how modified code is licensed are often referred to as ‘permissive’ licenses. While the original code covered by a permissive license stays under that permissive license, any modifications to it can be released under any license that the modifier chooses, open source or not. This means that permissively licensed code can form the basis of closed source products. Some argue that this makes permissive licenses more ‘free’ than copyleft licenses, in that people who modify the code are freer to choose what they do with it. Others argue that the lack of a requirement that modifications be open source means that permissive licenses are less ‘free’. As discussions of ideas of freedom - and particularly the idea of compulsion to be free - are essentially philosophical they fall outside the scope of this document. It is worth noting, however, that there are differing views within the free and open source world about what ‘free’ in the sense of freedom really means.

Example

As part of her astrophysics research Anne creates a standard for recording a specific kind of complex data object. She also writes code to create and parse files that adhere to the standard. Anne is keen that the standard becomes widely used, as minority standards are far less useful. Therefore she decides that she will

> release the code under a permissive license. Anne believes that by allowing creators of both closed and open source projects to use the same code to create and read the data objects, uptake and efficacy of the standard will both be helped along.

4.2.3 Strong And Weak Copyleft

Should you choose to include copyleft licensing conditions on reuse of your code, there is a further choice to be made. Copyleft licenses are broadly divided into two 'strengths': strong and weak. Strong copyleft conditions dictate that when a piece of software contains some of your code, the software as a whole must be distributed under your license, if it is distributed at all. The effect of this will be that the source code to all additions made to the code will be available.

Weak copyleft, on the other hand, means that when software contains some of your code, some parts of the software must be distributed under your license, if the software is distributed at all. Other parts may be distributed under other licenses, even though they form part of a work which is - as a whole - a modified version of your code. One effect of this will be that the source code to some additions made by others to your software may not be available as open source. Another effect may be that people may find it easier to 'productise' your code by adding closed

components and selling licenses to these closed parts.

To choose a specific weak copyleft license, you must also decide precisely which parts of your code will retain your license and what kinds of additions can bear a license of the modifier's choosing. This division can happen at one of three levels:

- Module level weak copyleft licenses dictate that each functional sub-section ('module') of code within the software is considered separately. Where a module contains some of your code, it must bear your license. Where it does not, the owner of that code gets to choose their own license for that module.
- File level weak copyleft licenses dictate that each collection of code and data that is distinct according to the computer's file system is considered separately. Where a file contains some of your code, it must bear your license. Where it does not, the owner of that code gets to choose their own license.
- Library interface level weak copyleft licenses are generally used where your code is a software library (a collection of software functionality which is usable by other programs via an agreed interface). Modifications of your library must bear your license, if they are distributed. Programs that use your library, and are perhaps distributed alongside it, need not use your license.

Example

Barry writes a piece of C code that examines websites over a network connection and generates image file diagrams of the links between their pages. Barry wants to release his code under a copyleft license because he wants to mandate access to the source code of all modifications of his program that are released. However, Barry also wants a wide variety of projects to be able to use his program - partly because he believes this will encourage the useful modifications and optimisations of his work that he wants to see. Barry decides to package his program as a library and use the library interface level weak copyleft license the GNU LGPLv2.1 on it, as it seems to him that this represents a good balance between encouraging reuse by both closed and open source projects while mandating the release of source of modifications to the parts he is interested in - his own code and the functionality it embodies.

4.3 Jurisdiction

A 'jurisdiction' refers to a specific location or territory and its system of law. Where a license specifies a jurisdiction, the licensor and the licensee(s) agree that the terms in the

license are to be understood in reference to that jurisdiction's law and that legal action resulting - for example - from breach of the license's terms will take place in that jurisdiction.

While jurisdiction is an important issue, it may be worth noting that traditionally free and open source software owners do not tend to seek monetary damages from those who infringe their licensing terms, but seek instead to compel the infringer to either abide by the terms or terminate their use of the code in question. For these purposes it is often unnecessary to resort to actually taking an infringer to court, particularly if they have a public profile and reputation to protect. Often simply requesting compliance can be effective, and if that fails, publicising the infringement can help achieve compliance. Not all free and open source software licenses specify a jurisdiction. In fact most are silent on the subject. In these cases any jurisdiction can be selected when and if necessary, although it is quite possible that the person you are trying to take to court will either ignore you or dispute your choice of location if it does not suit them. Finally some free and open source software licenses state that the jurisdiction is either up to the licensor or automatically that of the licensor (where they reside or principally do business).

Example

> CrabApple University is keen to write a policy on open source release, as an extremely large endowment specifies this as a pre-condition. Lawyers within Crabapple decide that a key piece of this policy must be the anointing of a specific license as the institution's preferred option when licensing out software as open source. At the first meeting to discuss which license to choose, it is pointed out that the institutional insurance policy does not cover legal action abroad. It is therefore decided that an essential feature of the anointed license must be that it enforces a local jurisdiction for any legal action resulting from the release of software it covers.

4.3.1 Patents

Do you or your institution own any software patents? If you do, and you release some code that embodies them under a free or open source software license, then you are very likely to be granting rights to use the relevant patent (in connection with that code) to anyone who chooses to use it - even if the license does not explicitly say so. In many jurisdictions, for example the UK and the US, licensing someone to take a particular action (like copying or adapting your code) can also impliedly license them to take all other steps necessary to take that action. These impliedly licensed steps would almost certainly include use of your

embodied software patent. It should be noted that people who adapt your code cannot expand its functionality to capture other software patents of yours - you grant rights only to the patents embodied in the code you released, not any subsequent form the code may take. Some free and open source software licenses say nothing on the subject of patent grants - although as noted this may not mean that they grant no patent rights. Some free and open source software licenses explicitly grant patent rights necessary to use, adapt and distribute the software.

Your free or open source software license can also include what is sometimes called a 'patent retaliation' clause. These sections of a license essentially say that anyone who brings legal action alleging that the licensed software embodies one of their software patents will lose the license you have granted to copy, use, adapt and distribute the code. Such a clause is intended to dissuade people from bringing this kind of legal action.

Example

Professor Dopaska has created a software-embodied process for predicting civil unrest that his university believes is patentable. The professor is keen to release the software embodiment of this process under the permissive Apache License, v2, as he wishes to see the process used ubiquitously to enhance his academic reputation. Knowledge transfer staff at Profes-

sor Dopaska's institution are not keen on this idea, as they wish to obtain the patent and build a spin-out company around it, and believe that the open source release will undermine the licensing income of this venture.

4.3.2 Enhanced Attribution

All free or open source software licenses specify that anyone who distributes or adapts the software must give credit to the original authors of the software somewhere in their distribution. Some free or open source software licenses go further than this, and specify that the credit must take a particular form and appear in specific instances, for example on the software's user interface every time it is run. This kind of stipulation is sometimes called 'enhanced attribution' or 'badgeware'.

Example

Edward creates a tool for visualising the frequency of occurrences of words in documents in an attractive way. Edward surmises that the tool will probably be used widely, but is not essential enough to most users' core business to support a paid licensing model. In truth, what Edward really wants to achieve is

publicity for the promotional consultancy he runs and for which the code was written. Edward decides to release his tool under theCommon Public Attribution License v1 as this mandates the display of a URL, a graphic and an attribution phrase every time the software is run. Edward provides his consultancy site's URL, logo and strapline along with the software so that users will help promote his business when they use and reuse his code.

4.3.3 The Privacy Loophole

If someone uses your code to create an online service or an in-house solution, perhaps adapting and improving it, most free and open source software licenses do not specify that the source to the adapted or improved version must be released. Most free and open source software licenses make it a condition of distribution that source code be released. In general neither making services available over a network, nor using the code, nor deploying the code within a single institution is defined as distribution within these licenses.

Some within the free and open source software community feel that this phenomenon, sometimes called the 'ASP (application service provider) loophole' or 'privacy loophole' needs to be fixed. Their argument is that fairness dictates that all those who benefit from the code must contribute their work back to the world, even if they are

not, strictly speaking, distributing the code.To address this issue, some free or open source software licenses make release of the source code a condition not only for distribution but also for internal deployment and/or making services available over a network using the software. These kinds of conditions are therefore particularly suited to code which is likely to be used in-house or as a basis for a networked service.

Example

Fareeda wants to build a business providing an easy way for users to build complex slide show music videos from their photographs on social networking sites. She finds a piece of open source software under the GNU Affero GPL v3 which provides a convenient way to handle the requirement that her site must interface with many external sites. Looking into the license, Fareeda discovers that its terms mandate that - if she modifies the code it covers and uses it as the basis of a service provided over a network as she intends - she must also make the source code to her modifications available to service users. Fareeda must now decide if her business model will be unaffected, aided or undermined by this potential responsibility.

4.3.4 No Promotion

Some free and open source software licenses explicitly forbid the use of the authors' names to promote a product or service based upon the authors' code.

Example

As part of a publicly-funded project Gerhentz University creates code which automatically adapts user interfaces to arbitrary display sizes. The institution is keen that the public funding leads to the maximum public benefit both in open and closed source software, and so wishes to release the code under a permissive open source license. However Gerhentz does not wish to be seen to be endorsing products that contain their code, as they feel that this may potentially damage the institution's reputation. Therefore it is decided that the BSD License will provide both the wide reusability and block on promotional use that they seek.

This chapter has presented some of the key differences between the various free and open source licenses, in order to help you consider what kind of license you might want to apply to your own code. Examples are provided to flesh out the kind of considerations that might feature in your own selection process.

You should note, however, that there is not a license for every possible combination of features, and so it may well be necessary to compromise on one or more category of features in order to actually choose a pre-existent license. To achieve this it can be helpful to rate the features you desire in order of importance.

There are online portals available for easily understand more about the open source licensing options like tldrlegal.com[1] or choosealicense.com[2]

[1] https://tldrlegal.com/

[2] http://choosealicense.com