Chapter 2
Free Software and Open Source Business Models

Arnulf Christl

Abstract This chapter discusses the nature of free and open source software from the perspective of business models that can be used to operate within this emerging industry. The focus in the chapter is on free and open source software in general rather than on the specific geospatial domain, however this area of activity is used in several places for reference and by example. The chapter also examines the closed and proprietary complements of the free and open source world and contrasts the rationale and behaviours of software developers and users within both market places. While it is not the intention to set one or the other markets up as a straw man, it is clear from the discussion that the free and open source alternatives have a number of advantages in terms of developing high quality outputs that are responsive to end user needs while embodying the principles of innovation and advancing knowledge.

2.1 Introduction

One of the first questions often asked of business people working in the free software industry is: "How do you make money if you give away the software for free?" Typically, this question is followed by an exclamation mark and a quizzical look in the eyes of the questioner that is meant to demonstrate that this is actually a very good question. In fact, this question is not hard to answer, although it is hard to convey a complete understanding of the answer to those working outside of the free software industry. In part, this is due to the perception of software as a product that is bought and owned, which is, itself, a concept that is learned by individuals at a very young age during basic socialization within a free market economy. However, there is more to software than just owning a copy of a product that can be legally used. Specifically, all software products have associated installation, customization, maintenance, training and upgrade needs, which vary considerably from product

Arnulf Christl
WhereGroup GmbH & Co. KG, Siemensstr. 8, 53121 Bonn, Germany; Open Source Geospatial Foundation, Beaverton, OR, USA, e-mail: arnulf@osgeo.org, arnulf.christl@wheregroup.com

G.B. Hall, M.G. Leahy (eds.), *Open Source Approaches in Spatial Data Handling*.
Advances in Geographic Information Science 2,

to product. This aspect of the computer software industry is substantially more complex that the selling of usage license fees, as consumers never actually buy a software product when they purchase it. Rather, they purchase a set of temporary usage permissions that must be agreed to before the product can be legally installed and used.

There are several reasons why it is important not to be impatient when people ask the above question. The most important is that often the starting point of understanding the free software industry is often ignorance, and this can only be overcome through education, removing ignorance and, even more importantly, removing false interpretations and twisted logic that result from applying the wrong concepts to the development and role of free software in a free market economy. The foremost of these false interpretations is to think that software is the same as any other physical good that can be traded. This chapter takes the view that this is not the case and develops an argument that supports the reasons why.

There are strong economic arguments that make corporations want consumers to believe that software is just another good. Terms like "commercial, off-the-shelf software" convey the character of a product. There are many more examples how to "personalize" software that is originally just a copy of a unique set of computer readable instructions. In some cases a unique (license) key is added to make a copy an individual, personalized entity. Sometimes software is even tied to hardware and whenever the hardware breaks or has to be exchanged the software will not run any more. All of this adds up to a fair amount of work that must be done to propagate and sustain the notion of software being just another good.

In addition to the Free Software industry, there is a growing community of developers who work within the Open Source community. Their predilection to produce and distribute source code for other developers to enhance has, in some cases, led to a neglect of understanding the basic underlying concepts that are required to make a living from using, deploying, developing, consulting, training and supporting software. Hence, this chapter deviates somewhat from the spatial data handling context of the remainder of this book and, instead, focuses on Free Software business models in general. Despite this, there is a clear spatial aspect of Free and Open Source Software (FOSS) that is embodied in the globalization mantra "Think Global, Act Local". While FOSS is being developed actively all over the world, there still is a need for direct, personal contact at the local level.

2.1.1 Disclaimer

This chapter describes FOSS as a legal concept on the one hand and as a development philosophy on the other hand. To show the advantages that FOSS offers to users, developers and businesses it is contrasted to proprietary models which have dominated the public view on the software industry for at least the past twenty years. Especially in the late 1990s proprietary software companies started actively to antagonize the FOSS movement because it endangered the proprietary business

model. This fight was and is supported by heavily funded campaigns and seemingly independent lobby organizations. The resultant spread of Fear, Uncertainty and Doubt (FUD) of the FOSS movement emerged partly out of ignorance of the advantages offered by alternative business, licensing and governance models, but it was also purposefully engineered to discredit FOSS and its development communities.

It is important to keep in mind in the following discussion that any licensing model of software is no proof of quality, regardless of whether it is free and open or closed and proprietary. To achieve good results with FOSS business models, it is not enough simply to apply an Open Source (OS) software license. Furthermore, not all software that is licensed as FOSS is automatically any better than proprietary software alternatives.

In February 2006 several leading FOSS communities bundled efforts to create the Open Source Geospatial Foundation (OSGeo) to support and build the highest-quality open source geospatial software tools possible. To achieve this, software projects go through an incubation process within this organization prior to their release and licensing (http://www.osgeo.org/incubator/process/process.html). Hence, there is a stamp of quality assurance that follows the same proven development and governance models that this chapter endorses. While this is only one approach to a development process that has business viability twinned with quality assurance and standards adherence, it is an approach that has achieved some success to date.

2.1.2 Business or Ethics and Both

There is a tradition to consider economics and ethics separately as though they are incompatible concepts. Unfortunately many commercial activities are unethical but completely legal, and sometimes the impression can arise that unethical but legal business is the most lucrative and expedient path toward commercial success. The seeming tensions between these concepts should be considered and a course charted that is ethical and legal as well as being able to foster business development. In this context, altruism is ethically high-valued and can therefore appear as remote from business activities. Because OS and especially the Free Software movement seem at first sight to be altruistic in intent, they are also regarded as being remote from viable business models. However, this reasoning is wrong, first because FOSS development does not have to be motivated by altruism, and second because business and ethics do not have to mutually exclude each other.

This chapter identifies a common ground between the two seemingly opposed perspectives and shows how to build a business with FOSS activities. The question of whether an enterprise is ethical or behaves in an ethical way can probably not be answered from a general viewpoint, and surely not in this short introduction to business around the FOSS model. Thus, the chapter tries not to judge business models but only to show which of them are viable in the long run.

2.2 Definitions of Software, Free Software and Open Source

This section provides an outline of the concepts described by the terms software, Open Source and Free Software. The discussion is only cursory, however it is necessary to go into some depth of understanding of the nature of software because it is fundamentally different from the physical products that consumers normally purchase. This discussion is necessary to gain an understanding of the business ideas behind associated FOSS models. As noted earlier, the discussion is general in nature and does not make a specific case for FOSS over other forms of software.

For practical reasons it is nowadays appropriate to abbreviate the activities of interest as FOSS. However, to explain the concepts behind FOSS it is helpful to consider the two aspects of FOSS separately. One is best described by the term OS as it focuses on the development methodology. The other is better described by the term Free Software (FS) as it regards the licensing model and legal background. These basic concepts are explained in the next sections.

2.2.1 Software and Hardware

To understand the nature of software it is helpful to discuss the ways in which it differs from hardware. There are more differences in the prefixes "soft" and "hard" than similarities in the suffix "ware".

Computer software cannot be executed (used) without hardware and vice versa. Hence, it is fairly common not to see the two terms as separate concepts. When the first computers were built the software was built alongside of them and there was no such thing as an IT industry that thrived on producing only software. The first computers consisted of tons of wires, bulbs and switches and were manufactured by scientists and electronic engineers (such as the Zuse Z11, which sold only 5 computers in 1956 – see http://www.epemag.com/zuse/part7c.htm). Every computer in that early era had its very own individual set of instructions.

Only much later, with the emergence of the personal computer in the early 1980s, was it possible to use one copy of software separately by one producer on the hardware of another (IBM was one of the major facilitators of this process). This was a sort of revolution that brought software its freedom of individual existence. However, this freedom did not last long because the newly emerged independent software allowed for a highly scalable business model. The task of implementing a software product only has to be funded once, but the results can be duplicated (copied) infinitely often at practically no additional cost as this does not require the same resources as the original product (essentially the creativity of the developer(s)).

To reduce the highly volatile and easily duplicable character of software it has become common practice to "bundle" software with hardware, such that if the hardware changes the software may stop running. The reason for this is not technical, as the above described independence of software from hardware has reached a very high level. Rather, the reason is that the license basically prevents changing the

configuration of the bundle. This confusion between hardware and software causes all kinds of trouble in understanding FOSS concepts that are focused explicitly only on software. However, hardware is a material concept, to which physical laws apply. Software, on the other hand, is immaterial and therefore physical laws do not apply.

A few examples that help to demonstrate this are shown in Table 2.1. Following the logic of this table, it is apparent that business models focusing on the availability of physical goods cannot apply naturally to software. Hence, the definition of what constitutes software in physical terms becomes difficult. To lessen the potential for confusion, software vendors have invented a new term, the "software product". This is essentially an attempt to make software resemble a physical good. Once software comes into existence it is naturally free of the limitations of physical availability (regardless of whether it is titled a "product" or not) since its supply is virtually endless at zero or a very small marginal cost. After the software has been implemented, all natural limits to its distribution are reduced to network connectivity costs or the trivial costs of the physical media that it is published on.

Many problems result from regarding software as just another good, or a form of solid ware, that is sold in boxes "off the shelf". As software has no physical representation and is made up of volatile bits and bytes, obviously other laws must apply. Perhaps it would be more appropriate to stop using the term "software" altogether and instead use something more appropriate like "nonware" or "technolyrics". This latter term nicely conveys two basic concepts inherent to software in that "techno" refers to the technicality of the (non)-"thing" at hand, while "lyrics" address the creative human component that the software is comprised of.

Table 2.1 Basic differences between hardware and software

Hardware	Software
If hardware (or any physical good) is sold the supplier suffers a loss of that good that can be compensated by payment.	If a copy of a software "product" is sold or given away, the original copy still exists. Hence, the supplier suffers no physical loss of the product because it is only a copy.
If hardware breaks or ceases to function beyond easy repair, it becomes useless.	Software cannot break in the same sense. A data carrier can be scratched (for example a CD) but the original of the software is not affected.
Hardware cannot be duplicated. Every copy needs the same amount of raw material and energy as any other. Copies of complex hardware will always be imperfect.	Software can be duplicated completely. Each successful copy of a software product is an identical reproduction of the original (the "raw material" is the source code, it does not run out).
Hardware can wear out, rust, or decay, and will eventually break and cease to function.	Software does not wear down, rust, decay or break. It may fall out of use, but it never loses its basic functionality.

2.2.2 The Early Days of Software Development

In the early days of computing, the original and natural way to solve computing problems was scientific collaboration. Every new software developed was based on prior knowledge and added some new aspect to it. At this point in time there was no need to use the term “Free Software”, since there was nothing to set this way of creating software apart from anything else.

Original (free) software development was only slowly displaced by proprietary thinking in the mid-1970s when it started to become economically viable to produce software that was independent of hardware. One document that shows this emerging line of thought incidentally is a letter sent by Bill Gates on February 3rd, 1976 (Gates, 1956). The content of this letter is void of the above inherent difference of software and hardware and confuses fairness with business models. In fact, the letter is not directed to the professionals of that time but to a group Gates describes as hobbyists. It turned out later that these “hobbyists” were in fact the “users” around whom Gates would create his business empire. In that same letter Gates asks: “Who cares if the people who worked on it get paid?” The obvious answer is that in the existing market economy model people need to take care of themselves and have to work out a way to get paid before they start to work. This is the basic difference between what makes a hobbyist a professional and it has nothing to do with fairness. The process of turning software into “property” was gradual and came hand-in-hand with a differentiation of hobbyists into users and developers.

2.2.3 The Open Source Development Model

Source code is the part of software that is human readable. It contains the information that is required to enable the software to run. All changes to software (removing errors, adding functionality, enhancements etc.) are done in the source code. Before software can be executed by a machine the source code has to be transformed (compiled) into binary or object code. This process is typically irreversible. Once software has been compiled it cannot be changed. Most end-user software nowadays comes in a compiled binary format and is shipped without the source code. Without the source code, software cannot be modified, repaired or enhanced and any control over what the software does is significantly restricted.

Open Source (OS) and the associated logo are trademarks of the Open Source Initiative (OSI) (http://www.opesource.org). Any software claiming to be OS has to abide by the terms and conditions defined by the OSI. These terms and conditions specify that the source code of open software must be published fully and without restrictions. Anybody can take OS software and look into its inner workings, change, improve and give away any number of copies to anyone else. Software is not seen as a secret hidden inside a black box, but as a living resource from which more and better software can be produced. OS licenses ensure that this concept is based on

a sound legal foundation. The legal background coincides almost completely with that of the FS definition which is described later in this chapter.

The other factors that make up a good OS project concern governance, communication and organization. This chapter only touches upon the topics that make the OS development model successful. A comprehensive set of instructions on how to produce OS has been compiled by Karl Fogel at http://producingoss.org, and this resource is an absolute prerequisite for anybody who wants to understand the implications of opensourcing (as a verb) their software. A lot of trouble can be eliminated if more people would educate themselves about these basic principles, including those who feel the need to argue for or against OS models. Two key factors emerge from Fogel's guidelines on OS development, namely "publish early and release often".

2.2.4 Publish Early

OS software is often started as a solution to a concrete problem. The sooner the solution is published the better for the project in the long run. This is crucial for all good OS software projects as it allows others to join the process at an early stage. If the solution is good then others may also be able to use it and start to contribute. Contributions can come in many different forms including actual software development as well as funding, documentation, testing, even publicity or recommendations of the software to other developers and users. The more people who pick up on the idea, the more testing, enhancements and development the solution will experience. Good communication right from the start is one of the crucial aspects of any OS project and sets it apart from closed development.

It can also happen that there already is a solution or other groups who intend to do the same things or have already progressed farther. In those cases it can save a lot of time and money not reinventing the wheel by joining efforts on a common project. It can also be an incentive to try and be better than the competing project, as diversity is good for any natural development. Moreover, the traditional grassroots process has a more spontaneous nature than a planned project that is organized from top to bottom with traditional power structures.

2.2.5 Release Often

Rapid prototyping and agile computing paradigms are followed by many projects. Publicly accessible code repositories allow developers and users alike to pick up the latest changes on a daily basis and keep up to date. Changes are documented in the repositories and distributed through special mailing lists so that anybody who is interested can follow edits in the code base very closely. Software in general is never finished and always buggy, and OS projects are no exception to this rule. However, there is a higher likelihood that bugs in OS software will be resolved

quickly and corrections to the software are typically added as patches that can be applied as regularly as desired. However, a proper software release is more than just the latest code from the repositories. It should be a tested, approved and as stable as is reasonably possible for public release. Release cycles should not be dictated by commercial considerations or marketing dates. Still, as a general rule, releases should come in a regular fashion and procrastination of release dates should be avoided as much as possible.

Basically, there are two different ways to determine release dates. One release type focuses on the availability of a defined set of new functionality that has been added to the software, or whenever it becomes awkward to stay up to date by applying a large number of patches at once. The other type defines an interval within which the next release is to be published. Depending on the amount of changes, the version number will then be incremented. For a collection of patches only the third version digit (the patch number) is increased. For changes or additions in functionality the second version digit is raised and the third set back to zero. Only deep functional changes, a complete re-engineering of the software and broken backward compatibility will cause the first version number to be changed.

Many of these development basics have also been picked up in slightly modified ways by large proprietary vendors who release patches and intermediary versions in regular intervals, sometimes even on a daily basis.

2.2.6 Free Software Licensing Model

The other factor that makes the OS approach to software development successful is the FS licensing model. It was developed in the early 1980s by Richard M. Stallman (http://www.gnu.org) in response to the growing possessiveness of businesses with respect to software developed by their employees. Stallman's mission was to use legal means to protect all intellectual work with a license that made sure that it could not be enshrined as individual property.

Stallman used a straightforward legal approach to achieve this. First, the work was put under copyright protection. This is the standard way to protect software or any creative work under both national and international copyright conventions (such as the international Berne Convention of September 9th, 1886 and its numerous revisions through to the present). Legal measures can be taken to protect the interests of the copyright holder. After having protected the software by a copyright covenant, distribution terms were added that made sure that everybody could use, copy, give away and modify the software – given that the changes are published using the same license. Thus, all software that ever appeared under this license would remain protected and tied to the same distribution terms.

Actually the requirement that all changes be tied to the same license was the most confrontational aspect of Stallman's license approach because it effectively excluded all proprietary business models that "protect" code by hiding it from the rest of the world. The most prominent example of this kind of license is the General

Public License (GPL) of the GNU project (http://www.gnu.org). This kind of license is sometimes called “viral” with reference to its potential to “infect” non-free software. In the same mindset it could also be seen as a vaccination for software to protect it from ever becoming proprietary. Another term frequently used to describe the effect of this license is “copyleft”. It is symbolized as a reversed c in a full circle like the copyright symbol, but mirrored. However, unlike the copyright symbol it has no legal meaning.

2.2.7 From Free Software to Open Source and Back

One of the pivotal developments in the conceptualization of FOSS was the publication in 2001 of Eric of Raymond’s book, “The Cathedral and the Bazaar” (http://catb.org/∼esr/writings/cathedral-bazaar/). In this book he summarized differences between a distributed, rapid development methodology and the traditional waterfall model. Specifically, he attributed the prior approach to OS and the latter to proprietary development. At that time it actually appeared as if the bazaar was the metaphor of the OS approach and that the only limitation for more commercial uptake was the software freedom mindset introduced by Stallman.

In another article, Raymond started an ongoing debate about the ambiguity of the term “Free” in “Free Software”, which in English language can be associated with “gratis” as easily as with “freedom” (http://catb.org/∼esr/open-source.html). In order not to discourage commercial businesses he proposed to start using the term “Open Source” instead of “Free Software”. This caused friction within the FOSS community finally leading to a schism that resulted in the creation of the OSI and long debates with Free Software Foundation (FSF) activists.

Bruce Perens, an early Debian GNU/Linux lead, was the primary author of the OS definition. This can be used to determine whether a software license can be considered to be OS or not, according to the following principles:

1. Free Redistribution: the software can be freely given away or sold. (This was intended to expand sharing and use of the software on a legal basis.)
2. Source Code: the source code must either be included or freely obtainable. (Without source code, making changes or modifications can be impossible.)
3. Derived Works: redistribution of modifications must be allowed. (To allow legal sharing and to permit new features or repairs.)
4. Integrity of the Author’s Source Code: licenses may require that modifications are redistributed only as patches.
5. No Discrimination Against Persons or Groups: no one can be locked out.
6. No Discrimination Against Fields of Endeavor: commercial users cannot be excluded.
7. Distribution of License: The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.

8. License Must Not Be Specific to a Product: the program cannot be licensed only as part of a larger distribution.
9. License Must Not Restrict Other Software: the license cannot insist that any other software it is distributed with must also be open source.
10. License Must Be Technology-Neutral: no click-wrap licenses or other medium-specific ways of accepting the license must be required.

The above terms of the definition of OS software are almost completely in line with the definition maintained by the Free Software Foundation (http://www.fsf.org/licensing/essays/free-sw.html). Hence, for all practical purposes Free and Open Source Software go together well. What makes the real difference between FOSS and proprietary software nowadays is the distribution terms, not the development model. Thus, the concept of FOSS is better conveyed by the term Free Software and its associated mindset.

2.2.8 Legalizing IT

The OSI lists 65 licenses (http://www.opensource.org/licenses/December 2007) that comply with the above terms and conditions and can thus legitimately use the OS trademark. These licenses cover a broad range from the restrictive Copyleft to more permissive ones that also allow the code to be used and distributed under proprietary terms and conditions. All of these licenses have gone through the OSI License Approval process (http://www.opensource.org/approval) and have a documented history of being applied to numerous OS software projects. Some of the licenses have also been approved in court in different national jurisdictions. This comprehensive repository is one of the references for the end user who wants to find out whether the software of choice is protected by a known and proven license.

Parallel to the OSI website, the FSF website lists 64 licenses (http://www.fsf.org/licensing/licenses/#SoftwareLicenses December 2007), many of which are also on the OSI list. The FSF adds a short description to each license identifying 30 that are compatible with the GNU GPL and 34 that qualify as Free Software licenses but conflict with the properties of the GNU GLP. These licenses have gone through the FSF Free Software Licensing and Compliance Lab which was informally established in the year 1992 and was formalized in 2001. From an end user's perspective all licenses are legally applicable. For developers, the FSF recommends to use only GNU compatible licenses as this allows developers to include and reference all software that is protected by the GNU GPL license (roughly three quarters of all OS projects).

These FOSS licenses are legal constructs that can be enforced in court. Hence, there is no difference between proprietary and FOSS licenses with respect to the legal power that can be exerted to enforce licensees to abide by the terms of the license. It is usually much harder for an individual (regardless of whether the individual is a user, developer or representing a business) to find independent information about proprietary license schemes and their applicability than for FOSS licenses.

Proprietary licenses are usually not developed through a consensus process and can be worded individually. Proprietary licenses can be and are changed at any time by the issuer. The above listed references to approved FOSS licenses have higher longevity, and are also a lot more stable and reliable.

The underlying general differences between proprietary and FOSS licenses becomes apparent when comparing license texts, especially with respect to the distribution terms. All proprietary licenses contain a lot of detail on what users are not allowed to do, whereas FOSS licenses explicitly grant users a variety of rights. These rights include the permission to copy, redistribute, re-engineer and modify the software or any parts of it, and, most importantly, to give away the software to be used by anybody else.

The restrictions that proprietary licenses have, in combination with the virtual and easily copyable nature of software in general, lead to a high potential of criminalization of the end users. FOSS licenses work the other way round by explicitly defining what users are allowed to do, especially to copy, change, modify and redistribute the source code and object code or any part of it. Hence, using FOSS licenses makes life easier for users and reduces the risk of breaching license terms and breaking the law.

2.3 Life Cycle Versus Development Cycle

Figure 2.1 shows a conventional software life cycle model compared to the OS development cycle model. The left hand side of the diagram shows some examples of how the life cycle model is implemented in a proprietary environment and reference to the right side shows how it compares to the OS development cycle.

Proprietary software vendors typically impose artificial life cycles on software to force users to upgrade to newer versions. In fact, some licensing schemes have an end date after which the software may not be used any more even if it would work appropriately. Users then have no other choice other than to upgrade to a newer version because the old version is no longer supported. Hence, it falls out of use and the "death" suggested by the term "life cycle" is caused by marketing mechanisms designed to keep the machine running.

Upgrading to new versions of software may require the payment of additional fees, often on a regular basis, by signing a software maintenance contract. Such a contract is intended to protect the user's initial investment in acquiring the usage license for a package. Without maintenance, the permission to use the software will end after a period that is defined by the license issuer. This effectively means that the initial cost of acquiring a usage license is lost. Hence, from this perspective the cost is not an investment but a one time expenditure. This issue is especially important in the spatial domain where there still is a heavy dependency between software and data in the sense that many vendors implement proprietary (closed) data formats that can only be read by the software they were created in.

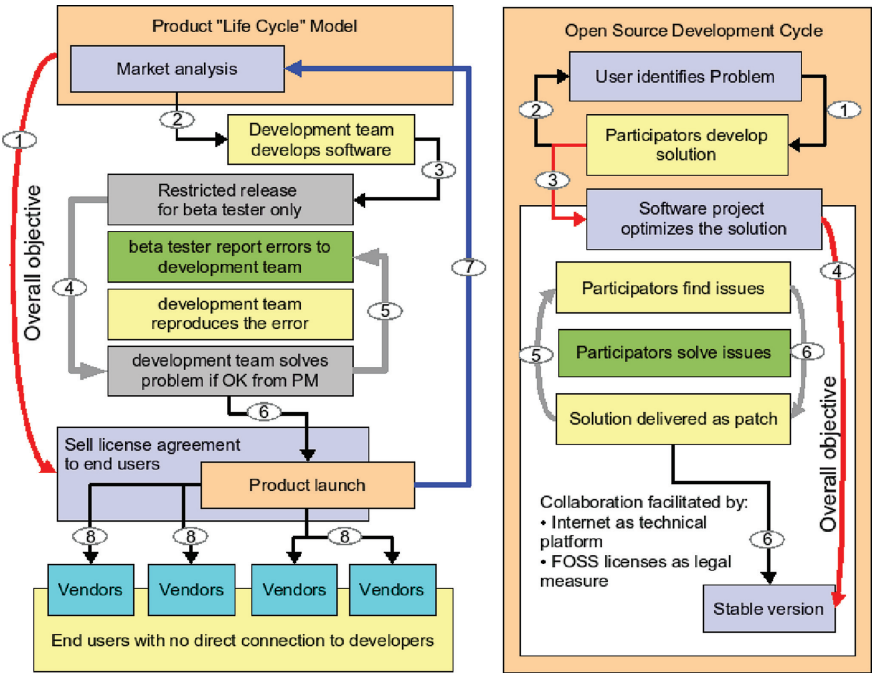

Fig. 2.1 Development cycles
(Source: Arnulf Christl 2007, http://www.mapbender.org/presentations/AGIT/modified)

Proprietary software users usually have no mechanism to influence the direction of development or new functionality that may be added to a package. This decision is taken in the best interest of the software vendor who may or may not take into account what the users want. Major releases often require migration of data, adjustment to new interfaces, and potentially migration of dependent software like applications running on top of a database. In contrast, OS environments allow for a faster iteration of development cycles because there is no profit-oriented marketing overhead that has different interests in where development goes.

The overall objective of most proprietary software vendors is to maximize profit (1) by selling as many license agreements to end users (8) as possible. The initial product design will be thoroughly influenced by a preceding market analysis that tries to predict the expected user behavior for the coming years.

After designing the "product" it goes into the planning and implementation phases (2). This is where core development takes place and it is not different to the FOSS development cycle. Test cycles start when the first beta version is ready for release. It can be distributed to selected beta testers who, in some cases, may have to pay for the privilege to be the first to be able to test the software. The hardware industry, as an example, has a vital interest in testing their components with the newest operating systems as soon as possible to be able to present compatible hardware when the software is released to the market.

The beta testing cycle iterates for a predefined time that is scheduled according to the marketing plan. Errors found during beta testing (4) may or may not be resolved (5). This is up to the product management team that controls the overall speed and path of development. Sometimes enhancements will not be implemented immediately but deferred for the next version to make long term maintenance contracts more attractive. There must be convincing arguments to make the user want to upgrade to the newest version.

At some point in time the stable version will be released. Usually the release date does not coincide with the software development being "finished". There are several reasons for this, the first being that software is always a work in progress and never really is finished. Another reason is that the introduction of new software or a major release change is a delicate undertaking. If it fails, the whole carefully planned schedule is in danger of failing. Depending on the type of software, releases will usually coincide with a major industry event, trade fairs or the period immediately prior to Christmas. The time to release proprietary software to the market is when end users are most receptive to chose it. There are plenty of examples where the schedule did not work out as planned, and this is sometimes reflected in the product name as a year number.

After launching the product vendors and resellers start to distribute the software (8) and the whole process iterates (7) as long as the market analysis promises high enough revenues for overall profit (1).

The overall objective (4) of the OS development cycle shown on the right hand side of Fig. 2.1 generally focuses on producing stable software that solves a problem. The cycle starts because someone has a problem (1) that can be solved by using software (2). If the solution is good it might attract more developers (3) and spawn a new project.

In the OS realm the terms "developer" and "end user" are much less clear cut than they are in a general sense. Every developer most of the time is also a user of other software, be it an email client, Web browser, operating system, or database. This is true regardless of whether the developer works in an open and free environment or in a proprietary environment. Users are the most important factor when testing software for usability and market acceptance. OS environments encourage communication between users and developers, greatly reducing time-to-market for new features and bug fixes. Because there is no explicit vendor role that divides developers from users they can interact more easily and participate in a collaborative work process. This justifies using the much more appropriate term "participant" for software developers and users alike.

A new software project will only come into existence and develop if enough people participate. Many OS software projects get born, wither and die because nobody uses them (6) and nobody has a reason to continue to develop them (5). Software stays in use as long as this cycle continues to iterate. In a proprietary environment this iteration can be kept going artificially for some time before the software actually dies. On the other hand, this iteration can also be stopped at will and at any time by the proprietary owner and copyright holder of the source code. This often happens in the course of software acquisitions by proprietary enterprises

who then discontinue development. In contrast, it is less likely for this to happen in the FOSS software development model because it is always possible to pick up the latest available version and continue development. This is also the reason why OS software is not threatened by corporate takeover or merger, and another reason for the difficulties that large enterprises with an exclusive proprietary business model are currently faced with.

2.4 FOSS Governance: Taking Back Control

The core of the proprietary business model is to maintain full control of software development and explicitly to limit the users' rights on the software. This means that users have no control over how software is developed, not to mention when and which new functionality is added or removed.

As noted earlier, a specific characteristic of the geospatial realm is the high dependency of software on data. Without geodata, geoprocessing software is rendered useless. A lot of spatial analysis needs to be able to process data across different domains and thus potentially across different software architectures. The need to be able to access datasets regardless of the hosting software environment is so strong that it is becoming more and more difficult for proprietary vendors to create typical vendor-lock-in situations by implementing closed formats.

FOSS governance models open up the development process by a group that takes decisions in an open and transparent way. There are many different governance models depending on the history and size of the project, the intended target audience and user groups. Small projects are often initiated by a single developer who is automatically in control of the code repository. As new developers join the project the initiator might choose to open up governance or keep it centralized. Quite a few projects can flourish perfectly well under a centralized model if the head is a good leader. One such project is the Geospatial Data Abstraction Library (GDAL http://www.gdal.org/) started by Frank Warmerdam in the late 1990s (see Chap. 5 of this text). As GDAL became more and more important to many other projects and businesses, its governance was opened and a project steering committee was created (currently with five members). This ensures that development of the project is not controlled by the interests of one individual.

Other projects such as Mapbender (http://www.mapbender.org/), initially started in 2001 as proprietary software that was implemented and owned by a single company, namely CCGIS, which was the forerunner of the Where Group (http://www.wheregroup.com). After releasing it as FS under the GNU GPL license in 2003 the development process gradually changed. A formal project steering committee was then created to allow enterprises and users who made heavy use of the software to join this group. When Mapbender became a formal OSGeo project it went through the OSGeo Incubation Process (http://community.osgeo.org/incubator/process/ process.html), which included a thorough revision and enhancement to the governance model. Currently, members representing companies, users and individual developers form the Mapbender steering committee (http://www.mapbender.org/index.php/PSC).

Independent of the governance model chosen for a FOSS project, the license terms always allow a project to be developed in a different direction than that planned by the group initially in control. This process can be detrimental to project development and can lead to the dilution of effort. One such example is the Java Unified Mapping Platform (JUMP) that is currently maintained by several different communities including:

- http://openjump.org/
- http://jump-project.org/
- http://www.vividsolutions.com/jump/
- http://www.saig.es/en/kosmo.php
- http://deegree.sourceforge.net/src/demos.html
- http://jump-pilot.sourceforge.net/.

To avoid this kind of dilution of resources, successful projects must put a lot of effort into meeting the needs of users, developers and businesses and to run and operate the project with an open and transparent governance style.

2.4.1 Decisions in a Voting Processes

A good example for an open, collaborative governance model is the MapServer project (see Chap. 4 of this text). It started as a research project that was later joined by a private company, DM-Solutions Group (http://www.dmsolutions.ca/), as well as individual contractors. The project is very successful and has grown a large community with very different needs. To be able to address better the broad spectrum of requirements, a formal entity was needed. The first official Request for Change (RFC 1, http://mapserver.gis.umn.edu/development/rfc/ms-rfc-1/) describes how this new MapServer Technical Steering Committee determines membership, and makes decisions on MapServer technical issues.

The voting system is very common throughout OS projects. In brief, the technical team votes on proposals on the developer mailing list server. Proposals are available for review for at least two days, and a single veto is sufficient to delay progress though ultimately a majority of members can pass a proposal. The following steps describe this process in detail:

1. Proposals are written up and submitted on the mapserver-dev mailing list for discussion and voting, by any interested party, not just committee members.
2. Proposals need to be available for review for at least two business days before a final decision can be made.
3. Respondents may vote "+1" to indicate support for the proposal and a willingness to support implementation.
4. Respondents may vote "−1" to veto a proposal, but must provide clear reasoning and alternate approaches to resolving the problem within the two days.
5. A vote of −0 indicates mild disagreement, but has no effect. A 0 indicates no opinion. A +0 indicate mild support, but has no effect.
6. Anyone may comment on proposals on the list, but only votes from members of the Technical Steering Committee will be counted.

7. A proposal will be accepted if it receives +2 (including the proposer) and no vetos (−1).
8. If a proposal is vetoed, and it cannot be revised to satisfy all parties, then it can be resubmitted for an override vote in which a majority of all eligible voters indicating +1 is sufficient to pass it. Note that this is a majority of all committee members, not just those who actively vote.
9. Upon completion of discussion and voting, the proposer should announce whether they are proceeding (proposal accepted) or are withdrawing their proposal (vetoed).
10. The Chair gets a vote.
11. The Chair is responsible for keeping track of who is a member of the Technical Steering Committee.
12. Addition and removal of members from the committee, as well as selection of a Chair should be handled as a proposal to the committee.
13. The Chair adjudicates in cases of disputes about voting.

Interaction and communication is achieved by using public mailing lists (http://lists.umn.edu/archives/mapserver-dev.html), forums, wikis, IRC chats, conferences and meetings. Especially, public mailing list archives contain a wealth of meta information about a project's past development, current health, and might even allow some predictions about its future. The main reason why many developers are not yet consciously involved in OS development methodologies is because they work in environments in which the underlying proprietary business model relies on software being treated like a scarcity, and its inner workings as a secret. Any public or open voting system would simply not work with this business model.

2.4.2 Limitations of Retail FOSS Business Models

Both OS and, especially, FS are frequently associated with a general anti-business ideology. From the limited perspective of a proprietary software vendor this might seem true because selling individual license usage agreements for software that is also available for download at essentially no cost at all is difficult, if not impossible.

In the early days of OS, business was recognized mainly as a gift economy (see Raymond, 2001), which did not scale well and lacked a solid foundation for enterprise businesses. Most information published around OS business models still focuses on the traditional approach in which an enterprise wants to make money by producing one particular ("their own") software. This only works for software that can be distributed in a retail style as a commodity allowing for high scalability and keeping the price per unit low. Each unit is an enabling technology that through its use will generate an added value that is higher than the individual cost, or else there would be no reason to purchase the software. The overall revenue generated by leveraging this enabling technology is a multiple of initial licensing costs, as Bruce Perens shows with the example of Microsoft:

> Microsoft is a tool-maker, and the effect of the tool-maker on the economy is tiny next to the economic effect of all of the people who are enabled by the maker's tools. The secondary economic effect caused by all of the people and businesses who use an enabling technology is greater than the primary economic effect of the dollars paid for that technology. And of course the same is true for Open Source software. Perens, Feb. 16 2005 (http://perens.com/Articles/Economic.html)

The results of this approach are recipes to make a profit by slightly bending Free and Open definitions or by combining them with proprietary models from the outset. Koenig (2004) identifies seven strategies to generate business and leverage OS:

- The Optimization Strategy
- The Dual License Strategy
- The Consulting Strategy
- The Subscription Strategy
- The Patronage Strategy
- The Hosted Strategy
- The Embedded Strategy.

These business strategies usually involve mixing proprietary and FOSS-based models, or are a vehicle to enhance business generated by selling hardware. They are hybrid business models derived from the needs of large enterprises and, for the most part, do not apply to small scale businesses or to highly specialized areas like the geospatial domain.

These niches are better addressed by small and medium-sized enterprise (SME) approaches and especially by pure service providers. From this perspective FOSS allows the adherence to much more attractive business models because it is a lot easier to provide services for a software without restrictions concerning its distribution than when distribution is restricted in some way. This distinction is probably the most important between proprietary (retail) and FOSS (services) business models. It also shows that the focus does not lie on one specific software but on the potential to use any software from the FOSS world to build specific solutions. However, this grassroots-oriented approach does not scale well for large enterprises.

Another reason for the general trend to associate FS with an anti-commercial attitude is that IT companies still rely heavily on proprietary business models and do not recognize the advantages offered by FOSS. Large companies with a long tradition of selling software usage licenses have an infrastructure that has been optimized mainly to market products. By releasing their software under a FOSS license they would destroy this source of income and would go out of business quickly. Therefore measures have to be taken to ensure that the proprietary business model will survive long enough to build a more reliable business model that can also take the specialties of FOSS into account.

Nonetheless some large enterprises have also started to embrace OS and use it to enhance their business portfolio. In many cases these efforts still ignore the fundamental aspects of FS licensing schemes and governance models and only try to make a profit from using OS (for example the GNU Linux operating system) instead of proprietary third party software (like the Microsoft Windows operating system).

Depending on the targeted market, services around software make up an increasingly larger share of potential revenue than income generated through collecting usage licenses fees. In the consumer market, scalability is so high that individual license costs can be kept at a relatively low level. Specialized software with smaller markets cause higher costs per installation making it more difficult to keep license fees on a low enough level not to scare away potential customers. Especially in the geospatial realm, additionally, a lot of professional expertise is required to get the software to run with the data and interact with other components. This is true regardless of the underlying software licensing scheme, and in this respect there are no inherent advantages for the application of either proprietary or FOSS licenses.

A further characteristic of spatial data is their longevity. Spatial data are persistent and do not easily migrate from one system to another. Often, the act of migrating data is a lot more expensive in terms of work load than what initial licensing costs plus long term maintenance costs of the underlying software add up to. The migration from proprietary to FOSS-based environments opens up a source of income focused on FOSS. The subsequent professional operation of the infrastructure is another traditional source of income that also applies for proprietary software.

2.4.3 Hybrid Proprietary/FOSS Business Models

Anybody can use FS to make a profit by using it for commercial activities or by providing services. As noted earlier, there are no legal complications in using FOSS licenses in a business or commercial context (http://www.ibm.com/developerworks/opensource/newto/#8). This also includes software that has been protected by Copyleft licenses (for example the GNU GPL). The only limitation is that it is not allowed to bundle and resell modified versions without opening the corresponding modified sources.

Large enterprises with a long history of selling software usage licenses have a difficulty in adopting FOSS business models because they operate complex software product design, marketing and distribution networks. The technical need has been rendered obsolete by the emergence of the Internet as a distribution platform. In the professional environment, the need for costly marketing campaigns is reduced due to most information being available ubiquitously via the Internet, newsgroups, domain communities and so on. In this regard the fact that FOSS licensing models have created new fields for commercial activities that are not focused on selling boxes but on providing services is a distinct advantage.

International Business Machines (IBM, http://www.ibm.com) has recognized this development and over the past years has developed a hybrid business model. IBM offers services for many OS software components and installs a growing share of hardware with the GNU Linux operating system, Apache webserver, MySQL database and PHP scripting language (LAMP) stack and distributes the Firefox Web browser, OpenOffice.org suite and so on with their retail hardware. The release of the integrated development environment Eclipse (http://www.eclipse.org/) as FS

further opens up the developer market. IBM can make revenue within this market as a proprietary software vendor with its specialized development extensions. Besides this proprietary business model, IBM is also the largest software patent holder worldwide.

The business model of Oracle Corporation (http://www.oracle.com/) focuses on selling proprietary software usage licenses and on dominating the market by acquisition of other companies. Nonetheless Oracle has started to develop an OS plan with the argument that this reduces costs and increases stability for its customers. The corresponding text on the companies Web page reads:

> Today, many customers are using Oracle together with open source technologies in mission-critical environments and are reaping the benefits of lower costs, easier manageability, higher availability, and reliability along with performance and scalability advantages. (http://www.oracle.com/technologies/open-source/index.html December 2007).

Applying these positive attributes to the competing OS database system PostgreSQL (http://www.postgresql.org/) shows how difficult it can be only to adopt a partial OS strategy.

Another Oracle strategy to address the growing OS economy is to address directly the developer and Internet Service Provider (ISP) markets. The release of the gratis version of Oracle (XE, express edition) is intended as an incentive for developers (OS and proprietary alike) to implement software that runs exclusively on top of the Oracle database. This will enlarge the overall market share for Oracle and bind new customers through vendor-lock-in. This kind of commercial activity does not leverage or support FOSS, neither does it add much to the revenue generated by selling licenses but it does help to grow market share.

The geographic information system (GIS) company ESRI Inc. also operates on a proprietary business model and generates revenue by selling software usage licenses and maintenance contracts. ESRI also supports the use of OS software components but almost exclusively from other domains and in limited ways, for example as an alternative to proprietary operating systems, software development environments or the Web server. Geospatial OS software, on the other hand, is in direct competition with ESRI's software and associated business model. Using the OS database PostgreSQL instead of the proprietary equivalent from the Oracle Corporation will reduce overall costs for the user. But at the same time it also opens up the potential of PostGIS (http://www.postgis.org) to ESRI users which might lead to fewer users favoring the proprietary equivalent offered by ESRI.

ESRI is also a shareholder of the company 52N (http://52north.org/) that has been founded to promote the conception, development and application of FOSS4G software for research, education, training and practical use. The governance model of the software released by 52N is still evolving and currently comprises a dual licensing scheme with the GNU GPL and a proprietary license. This allows members of the consortium to package the software together with proprietary components.

One outstanding example of a large proprietary enterprise in the geospatial realm that has gone partially OS with its own software is Autodesk Inc. It has released a completely rewritten version of its formerly proprietary software MapGuide as OS (http://www.osgeo.org/mapguide) and donated the code to OSGeo to underline its

commitment (see Chap. 7 of this text). In this context OSGeo serves as an independent legal entity to which community members can contribute code, funding and other resources, secure in the knowledge that their contributions will be maintained for public benefit.

Another OS initiative by Autodesk is the Feature Data Object (FDO) project (http://fdo.osgeo.org/). This is an application programming interface (API) for manipulating, defining and analyzing geospatial information for a variety of spatial data sources, where each provider typically supports a particular data format or data store. Here one of the strategic difficulties is caused by the strong ownership restrictions that Autodesk enforces for its proprietary computer assisted design (CAD) data formats and the growing need of the geospatial community to access, read and write CAD formats.

Marketing hybrid business models is very difficult because these models have to cover the gap between FOSS and proprietary sources of income. Most hybrid models only last for a transition period and have to undergo constant changes.

2.4.4 FOSS Services Business Models

Commercial activities around FOSS have a service character that is also applicable to most proprietary software. However, it is a lot easier to perform services using FOSS because license terms do not restrict distribution of the software. Additionally businesses explicitly focusing on leveraging FOSS will also help to further and support FOSS software development and the surrounding community. This is why hybrid models over time tend to erode the proprietary components. The main areas of FOSS business in the geospatial realm comprise:

- Education and Training
- Consultation
- Installation and Maintenance and Support
- Core development, implementation of new features.

Each of these services can be performed exclusively or in a value chain that covers all aspects from the inception of a software project through to long-term maintenance. FOSS can add value to all of these services instead of taking away potential sources of income as might appear at first sight. To complete this chapter, the following sections describe the detail of each service type and map it to real world examples in the geospatial FOSS domain.

2.5 Education and Training

Education and training are traditional ways to make a living within the knowledge domain. The basic idea with this form of activity does not really differ when training users of either proprietary or FOSS licensed software. The business model behind

training is straightforward. Participants pay a direct cost for the time that the trainer spends teaching the software.

Using proprietary software in training may raise costs due to additional license fees and reduce the potential income for the trainer and both the company providing the training and also the institute receiving the training. However, since training is a precondition to efficient use of software most proprietary vendors have a special educational licensing program that reduces these costs. Starting at elementary school level and going up to university and research institutes, proprietary software vendors grant usage licenses at reduced fees or for no cost at all.

The reasoning behind giving away software licenses to students for free is straightforward. Specifically, it allows them to become educated using a specific brand of software and whenever trainees complete their education and move into a position where they can influence decisions for or against a software package they will naturally tend to recommend the software they know best. This also justifies why proprietary vendors spend considerable effort and costs on the preparation of tutorials and related educational material.

This motivation is much less pronounced in the FOSS world as competition with other projects has less of an economic advantage. The drawback of this relaxation of the competitive edge is that considerably less effort (if any at all) goes into the preparation of training materials and coursework. One major critique of OS software has always been the lack of good documentation and training materials. Some voice the opinion that this is being done on purpose and is a covered up business model (Fotescu 2007). However, the reason is more likely to be embodied in the competing demands that are put on participants within the OS community.

The obvious solution to this deficiency is to apply the same collaborative principles that make OS work in general produce coursework, tutorials and documentation. However, unfortunately, even in education, proprietary mindsets have recently started to spread.

2.5.1 Creating Course Materials

During development of a software typically only basic technical documentation is created. Some development environments and languages render this kind of documentation automatically. In all cases it is created by experts (developers) and intended to be used by experts (other developers).

Universities are highly scalable resources of education, but there is little incentive for professors to organize their students to produce thorough documentation and tutorials for any software that they create or contribute to in their coursework as long as a well known set is already available from a proprietary vendor. In the meantime, professional trainers create documentation for software packages, but they will not do it for free. They either have to charge more money for individual classes to recover the cost of creating the documentation, or to have a budget to cover startup costs.

One way to create excellent course materials is to do it collaboratively. The documentation produced by the developers is the basis on which professional trainers can produce course material. This course material is published along with the software package and can be used, extended and enhanced for free. University courses are ideal places for thorough peer review, and enhancements can go back into the course material. During university courses, it is often required to create homework which can be integrated into the larger curriculum. Practical exercises and (scientifically) verified solutions round up the effort that collectively has the potential to create an excellent source of tried and tested documentation and practical exercises.

Professional trainers can supplement their practical exercises to improve the training materials with real-world aspects that would also help to put some reality into university-based curricula. Yet again, all of this is only possible if all documents created during these work flows are published under license with no restrictions in the distribution terms. Two of the better known examples are the Creative Commons license family and the GNU Free Documentation License (GNU FDL).

There are good examples in the geospatial world that demonstrate how this approach can work. The software project GRASS (http://www.osgeo.org/grass) has excellent educational resources due to its long term involvement with university education (see Chap. 9 of this text). Other projects that are more deeply embedded in the software architecture of many FOSS4G projects like PostGIS need more technical and less scholarly documentation and training materials. In this case the documentation is created and is maintained by the project itself (http://www.postgis.org/documentation/).

2.5.2 Audiences and Participants

There are three distinct audiences for training each with individual business opportunities, namely:

- Software developers, experts and trainers
- Professional users
- University and scholarly education
- Proprietary think trap.

The first group can make a living out of providing training, consultation, customization and development of software. Individuals in this audience are usually capable of processing all required information around a software package without the need for a trainer. Instead, their background knowledge and experience gives them the capability to acquire the knowledge about a new software package through self-training. Good software documentation will help them get started more easily. If the documentation is poor or locked away, the software will not be used or distributed as widely as it might be. The competitive advantage for the software project is that developers and experts act as multipliers.

The second group spends money. It is the audience that actually needs information and background to software packages to help them solve their problems. They might not have the expertise that developers have or lack the time to learn through use and experimentation. They are prepared to invest in learning how to use the software to solve explicit problems that they need to solve efficiently. If this comprises learning how to use the software within a short time frame through professional trainers, then they will adopt this approach. This type of individual or group training requires a trainer of a high level of proficiency, and justifies higher prices.

University and scholarly audiences comprise students and professors. Software itself is becoming an important asset for conveying information and helps students learn. Additionally, competence in software helps students to qualify for the job market upon graduation. However, professors need to invest time in software developments in order to keep up to date, otherwise the chances are high that they will not help disseminate new ideas and may paradoxically become obstructions to the process of knowledge growth and dissemination.

This kind of learning is slow but thorough. There is time to enhance details, ask critical questions beyond the daily chores of coding, and do research. During courses, written work and, increasingly, multi-media content are created. All of this work can become a valuable part of an overall software project if it is protected by a free and open license and published online. Technologies to do this are Wikis, collaborative books, content management systems, document management systems and so on.

The fourth audience is often explicitly targeted by proprietary vendors with special license rates. Vendors perceive this subvention as a long term investment. This is because, as previously mentioned, students who have learned how to use one package of software will tend to use that package when they start to work in the professional market.

2.5.3 Providing Teaching

More than anything, the costs of a trainer vary by location. A local trainer in Brazil will be able to offer much better conditions than somebody coming to Brazil from Germany, the United States or Australia. Simply not having to travel half way around the world will reduce the overall costs of training. Among the other benefits of operating locally, bills do not need to be issued across borders, taxation issues are easier, and people can meet more easily to prepare the teaching materials. Many local points of presence are required to offer teaching in native languages, hence this is a good opportunity for local service providers.

There are topics and software packages where the expertise initially lies with one or only a few individuals. This is the unique selling proposition (USP) that these individuals can exploit until the innovation has disseminated. The expertise is naturally first adopted by prior scholars. This competitive advantage allows an individual or a company to be one step ahead of the others, however it is lost or at

least reduced during training. Thus, it is necessary to stay ahead of the rest of the world, which is only possible by continually increasing capabilities and expertise.

Exploiting a single temporary advantage (leading know-how) is not a good foundation for a long-term business model. Rather, it is more sustainable in the long-term to maintain technical advantage through ongoing innovation and ingenuity, which is one of the driving factors in FOSS models.

Teaching or training can be as diverse as the requirements of the trainees. Depending on the type of or approach to training that is adopted, it can be useful either to implement a hands-on learning environment, lecture using static presentations, or work with a combination of both. In general, there are three distinct ways to educate in the use of software:

1. *Introductory information*. This type of teaching helps to orientate trainees, and provides a high level overview of relevant topics. It is usually conducted by individuals who are used to giving presentations focusing on one topic. Conferences are an ideal framework for this kind of teaching.
2. *Dedicated software courses*: This approach usually teaches trainees how to use one software package, or it has a distinct focus on one topic. This form of training is usually named after the software or topic. For example, in the case of "MapServer training" people will expect to learn how to use the MapServer software package. In the "Multi Band Satellite Imagery Analysis" course, people will expect to learn how to do just that, potentially with several different software packages.
3. *Problem-driven workshops*. This approach typically addresses one particular problem and ideally results in an ad hoc solution that precisely addresses the problem at hand. This type of training blends into consultation.

2.6 Consultation

Complex geospatial information, analysis and data acquisition frequently involve the use of several different software packages and the integration of different data sources. No single software package solves all problems. Increasingly, the term geospatial is taken to mean a way of looking at things with a perspective that takes location into account. Within this domain different software packages can, more often than not, be used to produce almost identical results with the same data. Sometimes it will be advantageous to use one package over the other. However, in many cases the evaluation process for selecting which package to use can be quite complex.

This is where professional consultation is important. This form of employment is possibly one of the best paid hourly rate jobs within the IT industry. The reason is that educated decisions have to be taken within a restricted time period and need to take a variety of factors into account. The education and training required to be able to make the right decision need the investment of a lot of time and money, as a wrong decision can have significant consequences for many years. Hence, good

consultation can be expensive, yet it is attractive from a business perspective both to the client and the consultant given its potential long term implications.

Taking software into operation for a customer involves considering many parameters. This is not special to FOSS but to software in organizations in general. The amount of money required for proprietary usage licenses usually only makes up a at most a third of what is needed for the overall effort required to install and maintain complex software systems. The cumulative costs resulting from using inappropriate software and the losses caused by unproductive employees over time can reach much higher levels than the initial acquisition cost of a proprietary license.

Vendor-lock-in can bind users to a specific software package for a long time, and prevent it from being substituted with other software, even if it becomes apparent that the system in use causes problems. Further, migration from one software system to another often raises numerous problems (e.g., converting data, changing workflows). Thus, consultation should always focus on using software that implements open standards and can be used easily in conjunction with software from other sources (see Chap. 1). Another advantage of the FOSS approach is its transparency and the possibility to start using it without initial expense.

2.6.1 Installation, Maintenance and Support

Installation of desktop applications can be straightforward, but in some instances they can also be somewhat daunting, especially if the application is unpackaged or if the application does not comply with the IT-security policies of an existing environment. Depending on the type of software and system environment under consideration, business models based on performing installations can be very different. For example, for server software it may be necessary to compile special versions that fit neatly into the customer's existing environment.

The maintenance of operating environments can be performed by client staff (who need to be educated and trained), but increasingly this level of maintenance is also outsourced to service providers. The complexity of geospatial data and processing in part is due to the niche character of the software. There is little or no scalability potential in spatial data infrastructures as each focuses on specific domains of interest. The emerging spatial software stack of OSGeo can be used to build a foundation for geoprocessing capabilities but the interaction of the components first has to be designed and then implemented within the existing architecture. This level of implementation and its maintenance typically require a relatively high level of knowledge, and support contracts can help to consolidate an infrastructure as well as reduce the requirement for deep internal knowledge among clients.

Support contracts are more flexible than pure maintenance contracts in that they can be used to maintain the system or to enhance features or extend functionality. Much of what is typically provided as first level support can be obtained through mailing lists for free but this cannot be fixed in a formal contract. Second level

support requires more knowledge of the application of the software in use and is requested by integrators or other service providers, who either provide first level support themselves, or are bound to provide guarantees which need to be covered by back-to-back agreements.

Third level support needs intricate knowledge of the inner workings of software and is often provided by the core development teams of software projects. This level of support can also help to leverage development of the software in the direction that is needed by the customer.

The business model is, again, straightforward. The client reports a problem that is solved by the service provider. Agreements to limit excessive support hours can vary from contract to contract as there are no general rules as to how these work. In many cases clients buy a support quota that is worked off by the service provider on request.

2.6.2 Software Development

Software development is the core of what is commonly associated with FOSS approaches. This aspect of work is probably the most difficult activity to make a living from. This is at least true for software developed in the FOSS4G realm, as it is a niche activity rather than a commodity. The scalable commodity type of spatial applications has been picked up amazingly fast by large Internet companies leaving little room for new software, but opening up a whole new secondary market of application mashups that are based on the new geospatial commodities.

There are different development methodologies that vary depending on what language the software is implemented in, and what kind of solution is addressed. Many FOSS projects start as any grassroots movement, namely as modest contributions of coding that solve specific problems of relatively limited scope. If such software solves a problem and people use it, then it has the potential to improve and grow. Once it has reached a certain level of acceptance, or is being used in professional contexts, chances are high that it will be further improved and consolidated. Over time, the software and the people involved in implementing it will mature, and development methods will accordingly professionalize. If the software is a complex package or involves many dependencies, sooner or later it will be necessary to organize it in order for the software to be sustainable in the long-run. The job of organizing and coordinating good development is usually not visible on the outside (the user's perspective) of a software package. Thus, it is hard to argue for money to support the development effort.

Generic code that can be used for many purposes and does not address only one need or solve one problem is the most difficult aspect of development to find funding for. Usually, this kind of development has to be cross-financed from implementing features that are often transparent to end users. The only way to argue for a generic approach is to be completely transparent about the required development, and to explain why the resources invested in developing generic software will pay off in the

long run. This is also a task that usually cannot be performed by a single developer. Rather, it needs project-level organization and, at best, an independent contact point for the project. In the FOSS4G world, this task can be taken on by a project or technical steering committee, that consists of a group of developers and power users who share the responsibility to "run" the project.

The independent contact point can also be a professional who problem solves for a fee (for example by making a support contract). In many cases the solution provider will be part of the development group of that software, but this is not a requirement. The ubiquitous availability and unconditional accessibility of FOSS allows anybody to enhance, repair and improve it, for money or out of any other motivation they may have (personal gratification in solving a hard-to-solve problem, peer accolades etc.).

In the FOSS community, depending on the license and the development contract, the implementer should or even must give the enhancement back to the rest of the world unencumbered by proprietary licenses. This is also in the interest of the customer because it enhances the chance that his/her extension will become part of the main software, and thus be available along with updated versions of the main software without additional implementation effort.

If the source code is locked away as-is in a proprietary environment, there is a natural monopoly that accrues to people who have access to the source code, hence those who can will contribute to its development. This inevitably results in less diversity and it ignores the potential of achieving the highest quality by opening up the code to the scrutiny of the maximum number of peer reviewers. Thus, the chance to implement longevity and robustness is foresaken.

As a client it is easy to adopt the quickest and (from a short-term perspective) cheapest solution. This will invariably be a hack. If the problem can be solved with a hack and afterwards the implementation falls out of use (e.g., one-time converter software), then this is a perfectly appropriate way to operate. Even then it makes sense to publish the snippets and fragments of code that led to that solution, enabling others to profit from the work that has already been done. To stay with the example of the converter, it will help people to convert their data into the new format more easily, which might produce follow-ups on the software around the new format. One such example noted earlier is the GDAL project which has a long history of collaborative development and gives generic access to a broad range of different formats.

The project hack appears to be the better solution for the short-term from the viewpoint of the customer and end-user because it is cheaper and addresses the immediate problem directly. However, in most cases it will pay off in the long-term to implement a generic solution because of the above arguments. This insight often reveals itself to the customer only after he/she has encountered a few painfully hard brick walls of frustration. It can be difficult to lend somebody who has gone through this frustration a helping hand and still insist on FOSS being the better concept. Thus, it is very important first to educate the customer of the implications of a specific strategy, and then provide good consultation evaluating the advantages of each implementation option.

2.7 Conclusion

This chapter has provided a review of free and open source business models and contrasted them against their counterparts from the closed and proprietary world of software development. The chapter started by providing definitions of software, free software and open source as a backdrop to the subsequent discussion. A differentiation was made between hardware and software as this is fundamental to the differing business models that can be adopted. The OS development model was then discussed and several of its underlying premises were noted. This was followed by consideration of the Free Software licensing model, and the relationship between the two was discussed. The body of the chapter examined the life cycle and development cycle of free and open source software and proprietary software, and concluded with a comprehensive review of the business orientation, rationale and limitations of the free and open source software community. In this discussion the education and training markets and the role of consultant services were given special consideration.

References

Christl A (2007) <http://www.mapbender.org/presentations/AGIT/modified>

Fotescu R (2007) The sorry state of the Open Source today, <http://beranger.org/feature/sorryfeature.php> 28 July 2008

Gates W (1976) Open letter to hobbyists, Homebrew Computer Club Newsletter, 2, 1, p.2

Koenig J (2004) Open Source Business Strategies <http://kp.cospa-project.org/retrieve/1537/opensourcebusinessmodels.pdf> 29 July 2008

Perens, Brue (2005) <http://perens.com/Articles/Economic.html>

Raymond E (2001) The Cathedral and the Bazaar: musings on Linux and Open Source by an accidental revolutionary, O'Reilly Media Inc., Sebastapol, Ca.