Facultad de Ciencias Exactas
UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES

INGLÉS/ INGLÉS II

TRABAJO PRÁCTICO Nº 6

TUPAR, TUDAI E INGENIERÍA DE SISTEMAS

➢ ACTIVIDAD DE PRELECTURA

1- El texto fuente menciona los siguientes criterios de selección. ¿Qué comprenderían según su parecer?

community- release- longevity- license- support- documentation- security- functionality – integration

2- ¿Qué criterio/ s considera más relevantes a la hora de elegir un paquete de software?

➢ ACTIVIDADES DE LECTURA

1- ¿En qué estudios previos se basó la autora para realizar este trabajo? **RESPONDA** brevemente.

2- **EXPLIQUE** la siguiente afirmación: "***The open source software market is in some ways very different from the traditional software market***."

3- **EXPLIQUE** por qué los autores afirman que el factor `comunidad` es el más importante al evaluar `*software libre*.`

4- **LEA** la sección `*Release Activity.*` **CONSIGNE VERDADERO** o **FALSO**. **JUSTIFIQUE** las falsas.

a- ……. Para evaluar la actividad de un proyecto se tienen en cuenta principalmente dos factores.

b- ……. El proceso de reactualización puede evaluarse al observar el historial de los cambios introducidos en cada proyecto.

c- ……. No puede evaluarse con qué seriedad se incorporan cambios en este tipo de proyectos.

d- ……. En general, se suelen lanzar versiones una o dos veces al año.

e- ……. Es más conveniente para el usuario que existan pocas y eficientes versiones a que existan muchas poco operativas.

5- **LEA** el apartado `*Longevity*.` ¿Acuerda o no con las siguientes afirmaciones?

a- *Age is not always a guarantee of survival*. **FUNDAMENTE.**

b- *The fast progression of the version number might be used to create a false sense of progress*. **FUNDAMENTE.**

c- *One thing that needs to be taken into account when products are not very young is whether or not there is still an active community around it.* **FUNDAMENTE.**

6- **RELACIONE** los conceptos *`copyleft` y `copyright.`*

7- **LEA** el apartado *Support*. **DESARROLLE** brevemente estas afirmaciones:

- Los dos tipos de asistencia existentes se pueden mezclar porque……

……..

- El término `*third party support*` alude a ……

……

- El hecho de que existan asistencias pagas significa……

……

- En el área de asistencia, la comunidad es muy importante porque……..

……

- Cuando los `*bug trackers*` encuentran un error, deben……
- Una de las formas en que se demuestra la madurez del código abierto es ……

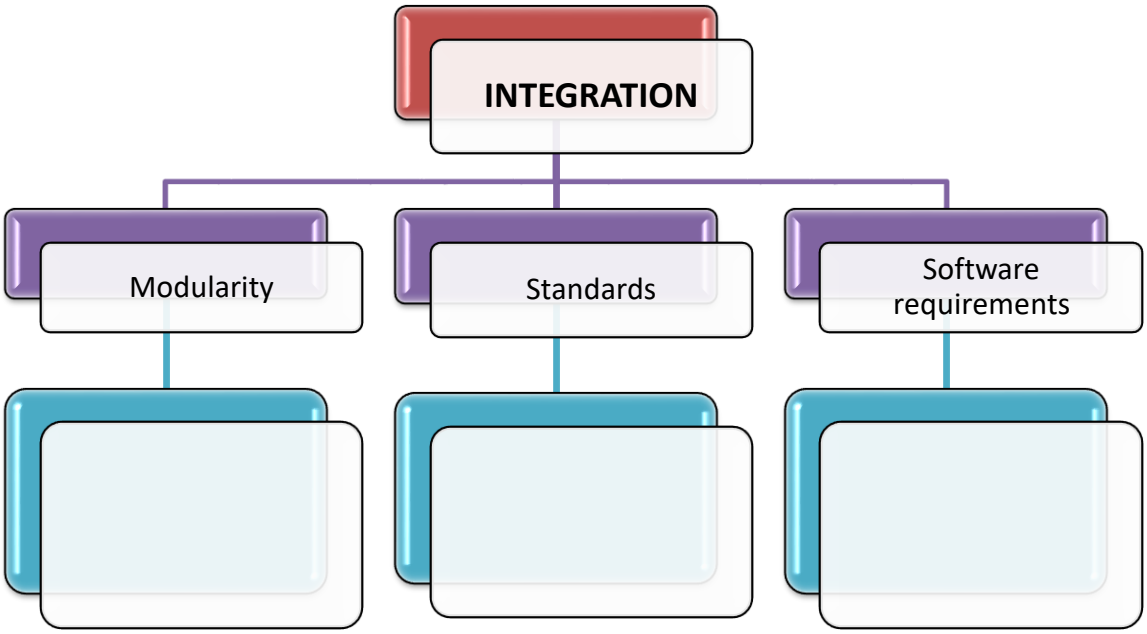
8- **LEA** el apartado `*Documentation*.` **COMPLETE** la siguiente tabla. **CARACTERICE** cada apartado.

Documentación del usuario	Documentación del desarrollador	Documentación de asistencia

9- **LEA** el apartado *Security* y **RESPONDA**:
   a- ¿De qué depende la seguridad del software?
   b- ¿Qué debe hacer un desarrollador si encuentra alguna vulnerabilidad?
   c- ¿Qué ejemplos sobre advertencias de seguridad se mencionan?

10- En la sección `*Functionality,*` se encuentra la frase: "release early and often". **EXPLIQUE** a qué se refiere.

11- **COMPLETE** la red conceptual con <u>frases</u> relacionadas a cada criterio en el área *Integration*.



12- **REDACTE** para cada apartado una idea primaria y una secundaria**.** Luego, **UTILÍCELAS** para redactar un resumen.

COMMUNITY	***El factor comunidad es el aspecto más importante al valorar un paquete de software.***
RELEASE	
LONGEVITY	
LICENSE	
SUPPORT	
DOCUMENTATION	
SECURITY	
FUNCTIONALITY	
INTEGRATION	

➢ ACTIVIDADES DE POST- LECTURA

13- Con la información del cuadro anterior, **REDACTE** un resumen. **UTILICE** el comienzo sugerido.

En la actualidad, las empresas y los usuarios pueden elegir qué tipo de software desean adquirir. Para ello, es indispensable evaluar las diferentes opciones del mercado teniendo en cuenta diferentes criterios. ……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

……

Chapter 1.6
Open Source Software Evaluation

Karin van den Berg
FreelancePHP, The Netherlands

INTRODUCTION

The open source software market is growing. Corporations large and small are investing in open source software. With this growth comes a need to evaluate this software. Enterprises need something substantial to base their decisions on when selecting a product. More and more literature is being written on the subject, and more will be written in the near future.

This chapter gives an overview of the available open source evaluation models and articles, which is compounded in a list of unique characteristics of open source. These characteristics can be used when evaluating this type of software. For a more in-depth review of this literature and the characteristics, as well as a case study using this information, see van den Berg (2005).

OPEN SOURCE SOFTWARE EVALUATION LITERATURE

The name already tells us something. Open source software is open—not only free to use but free to change. Developers are encouraged to participate in the software's community. Because of this unique process, the openness of it all, there is

far more information available on an open source software package and its development process. This information can be used to get a well-rounded impression of the software. In this chapter we will see how this can be done.

Though the concept of open source (or free software) is hardly new, the software has only in recent years reached the general commercial and private user. The concept of open source evaluation is therefore still rather new. There are a few articles and models on the subject, however, which we will introduce here and discuss more thoroughly in the next section.

Open Source Maturity Models

Two maturity models have been developed specifically for open source software.

The first is the Capgemini Expert Letter open source maturity model (Duijnhouwer & Widdows, 2003). The model "allows you to determine if or which open source product is suitable using just seven clear steps." Duijnhouwer and Widdows first explain the usefulness of a maturity model, then discuss open source product indicators and use these in the model. The model steps start with product research and rough selection, then uses the product indicators to score the product and determine the importance of the indicators, combining these to make scorecards. Finally it ends with evaluation.

Second, there is the Navica open source maturity model, which is used in the book *Succeeding with Open Source* (Golden, 2005). This model uses six product elements in three phases: assessing element maturity, assigning weight factors, and calculating the product maturity score.

Open Source Software Evaluation Articles

Aside from the two models, a number of articles on open source software evaluation have been written.

Crowston et al. (2003) and Crowston, Annabi, Howison, and Masango (2004) have published articles in the process of researching open source software success factors. In these articles, they attempt to determine which factors contribute to the success of open source software packages.

Wheeler's (n.d.) *How to Evaluate Open Source/Free Software (OSS/FS) Programs* defines a number of criteria to use in the evaluation of open source software, as well as a description of the recommended process of evaluation. Wheeler continues to update this online article to include relevant new information.

Another article defining evaluation criteria for open source software is *Ten Rules for Evaluating Open Source Software* (Donham, 2004). This is a point-of-view paper from Collaborative Consulting, providing 10 guidelines for evaluating open source software.

Finally, Nijdam (2003), in a Dutch article entitled "Vijf Adviezen voor Selectie van OSS-Componenten" ("Five Recommendations for Selection of OSS Components"), gives recommendations based on his own experience with selecting an open source system.

Literature Summary

Table 1 summarizes the criteria derived from the literature mentioned in the previous two sections and how they are discussed.

EVALUATING OPEN SOURCE SOFTWARE

The open source software market is in some ways very different from the traditional software market. One of the differences is that there is an abundance of information available concerning the software and its development process that is in most cases not available for traditional software.

The evaluation of traditional software is usually focused on the functionality and license cost of the software. In the open source world, the

Table 1.

Criterion	Duijnhouwer and Widdows (2003)	Golden (2005)	Crowston et al. (2004)	Wheeler, (2005)	Donham (2004)	Nijdam (2003)
Community	Y	Y	Team size and activity level	In support	-	Active groups
Release Activity	-	Activity level	Activity level	Maintenance	-	Active groups
Longevity	Age	Y	-	Y	Maturity	Version
License	Y	In risk	-	Y	Y	Y
Support	Y	Y	-	Y	Y	-
Documentation	In ease of deployment	Y	-	In support	Y	-
Security	Y	In risk	-	Y	Y	-
Functionality	Features in time	Y	-	Y	Y	Y
Integration	Y	Y	-	In functionality	In infrastructure	-

evaluation includes information from a number of other resources, giving a well-rounded picture of the software, its development, and its future prospects.

Using the existing evaluation models and articles discussed in the previous section, an overview is given here of the characteristics of open source software relevant to software evaluation and the information available on an open source software project concerning these characteristics.

Community

According to Golden (2005, p. 21), "One of the most important aspects of open source is the community."

The user community for most open source projects is the largest resource available. The community provides developers, user feedback, and ideas, and drives the project team. An active community helps the project move forward. It also shows the level of interest in the project, which can provide a measurement of quality and compliance with user requirements. A well-provided-for community also shows the team's interest in the user, allows the user to participate, and gives voice to the user's wishes and requirements.

The user community of an open source project consists of the people that use the software and participate in some way, from answering user questions to reporting bugs and feature requests. Users in the community sometimes cross the line into the developer community, which is often a line made very thin by encouraging participation and making the developer community accessible to anyone who is interested. In some cases, the user and developer community interact fully in the same discussion areas.

The community of an open source project is very important because it is the community that does most of the testing and provides quality feedback. Instead of using financial resources to put the software through extensive testing and quality assurance (QA), like a proprietary vendor will do, the open source projects have the community as a resource. The more people that are interested in a project, the more likely it is that it will be active and keep going. A large and active community says something about the acceptance of the software. If the software was not good enough to use, there would not be so many people who cared about its development (Duijnhouwer & Widdows, 2003).

The community is mostly visible in terms of the following (Crowston et al., 2004; Duijnhouwer & Widdows, 2003; Golden, 2005; Nijdam, 2003):

- **Posts:** Number of posts per period and number of topics
- **Users:** Number of users and the user-developer ratio in terms of the number of people and number of posts; if only users post, the developers are not as involved as they should be
- **Response time:** If and how soon user questions are answered
- **Quality:** The quality of posts and replies; are questions answered to the point, and are the answers very short or more elaborate? Is there much discussion about changes and feature additions?
- **Friendliness:** How friendly members are toward each other, especially to newcomers, also known as "newbies"; the community should have an open feel to it, encouraging people to participate

The depth of conversations, as mentioned in the fourth item, gives a good impression of how involved the community is with the ongoing development of the project. Much discussion about the software, in a friendly and constructive manner, encourages the developers to enhance the software further. The community activity is also reflected in other areas such as support and documentation.

Release Activity

The activity level of a project consists of the community activity and the development activity. The community was discussed above. The development activity is reflected in two parts:

- The developer's participation in the community
- The development itself—writing or changing the source code

The latter activity is visible mostly in the release activity. All software projects release new versions after a period of time. The number of releases per period and their significance, meaning how large the changes are per release (i.e., are there feature additions or just bug fixes in the release), illustrates the progress made by the developers. This gives a good indication of how seriously the developers are working on the software.

The open source repositories SourceForge[1] and FreshMeat[2], where project members can

share files with the public, provide information that could be useful to evaluate the release activity (Wheeler, n.d.).

An open source project often has different types of releases:

- **Stable releases:** These are the most important type for the end user. They are the versions of software that are deemed suitable for production use with minimal risk of failure.
- **Development versions:** These can have different forms, such as beta, daily builds, or CVS (Concurrent Version System) versions, each more up to date with the latest changes. These versions are usually said to be used "at your own risk" and are not meant for production use because there is a higher possibility of errors. A project that releases new versions of software usually publishes release notes along with the download that list all the changes made in the software since the previous release. Other than the release notes, the project might also have a road map, which usually shows what goals the developers have, how much of these goals are completed, and when the deadline or estimated delivery date is for each goal. Checking how the developers keep up with

this road map shows something about how well the development team can keep to a schedule.

Though a project might stabilise over time as it is completed, no project should be completely static. It is important that it is maintained and will remain maintained in the future (Wheeler, n.d.).

The project's change log can give the following information (Chavan, 2005):

- **The number of releases made per period of time:** Most projects will make several releases in a year, sometimes once or twice a month. A year is usually a good period in which to count the releases.
- **The significance of each release:** The change log or release notes explain what has changed in the release. These descriptions are sometimes very elaborate, where every little detail is described, and sometimes very short, where just large changes are listed. A good distinction to make is whether the release only contains bug fixes or also contains enhancements to features or completely new features. One thing to keep in mind here is that fewer, more significant releases is in most cases better than a large number of less significant releases leading to the same amount of change over time since the users will have to upgrade to new versions each time a release is made, which is not very user friendly. There should be a good balance between the number of releases and the releases' significance. If the project is listed on SourceForge and/or FreshMeat, some of the release activity information is available there.

Longevity

The longevity of a product is a measure of how long it has been around. It says something about a project's stability and chance of survival. A project that is just starting is usually still full of bugs (Golden, 2005). The older a project, the less likely the developers will suddenly stop (Duijnhouwer & Widdows, 2003). However, age is not always a guarantee of survival. First of all, very old software may be stuck on old technologies and methods, from which the only escape is to completely start over. Some software has already successfully gone through such a cycle, which is a good sign in terms of maturity. One thing that needs to be taken into account when products are not very young is whether or not there is still an active community around it.

The age and activity level of a project are often related. Young projects often have a higher activity level than older ones because once a project has stabilised and is satisfactory to most users, the discussions are less frequent and releases are smaller, containing mostly bug and security fixes. This does not mean that the activity should ever be slim to none. As mentioned before, no project is ever static (Wheeler, n.d.). There is always something that still needs to be done.

Longevity is checked using the following criteria (Golden, 2005; Nijdam, 2003):

- **Age of the product:** The date of the first release
- **Version number:** A 0.x number usually means the developers do not think the software is complete or ready for production use at this time.

If the project is very old, it is worthwhile to check if it has gone through a cycle of redesign, or if it is currently having problems with new technology.

Keep in mind that the version number does not always tell the whole story. Some projects might go from 1.0 to 2.0 with the same amount of change that another project has to go from 1.0 to 1.1. The fast progression of the version number might be used to create a false sense of progress.

Other software products are still in a 0.x version even after a long time and after they are proved suitable for production use (Nijdam, 2003).

License

The licenses in the open source world reflect something of the culture. The most important term in this context is "copyleft," introduced by Richard Stallman, which means that the copyright is used to ensure free software and free derivative works based on the software (Weber, 2004). In essence, a copyleft license obligates anyone who redistributes software under that license in any way or form to also keep the code and any derivative code under the license, thus making any derivatives open source as well.

The most well-known example of a copyleft license is the GNU GPL (General Public License; Weber, 2004). This is also one of the most used licenses. On SourceForge, a large open source public repository where over 62,000 projects reside, almost 70%[3] of projects use the GNU GPL as their license. There are some large and well-known products that do not use SourceForge, and some of these have their own license, such as Apache, PHP, and Mozilla (Open Source Initiative [OSI], 2005).

Because copyleft in the GNU GPL is very strong, an additional version was made called the LGPL (library GPL, also known as lesser GPL), which is less restrictive in its copyleft statements, allowing libraries to be used in other applications without the need to distribute the source code (Weber).

A non-copyleft license that is much heard of is the BSD (Berkeley source distribution) license. It has been the subject of much controversy and has had different versions because of that. Components that are licensed under the BSD are used in several commercial software applications, among which are Microsoft products and Mac OS X (Wikipedia, 2005a). The license of the software in use can have unwanted consequences depending on the goal of the use. If the user plans to alter and redistribute the software in some way but does not want to distribute the source code, a copyleft license is not suitable. In most cases, however, the user will probably just want to use the software, perhaps alter it to the environment somewhat, but not sell it. In that case, the license itself should at least be OSI approved and preferably well known. The license should fit with the intended software use.

As just mentioned, the license should preferably be an OSI-approved license. If it uses one of the public licenses, the better known the license, the more can be found on its use and potential issues (Wheeler, n.d.).

Support

There are two types of support for a software product:

- **Usage support:** The answering of questions on the installation and use of the software
- **Failure support or maintenance:** The solving of problems in the software

Often, the two get mixed at some level because users do not always know the right way to use the product. Their support request will start as a problem report and later becomes part of usage support (Golden, 2005).

The way support is handled is a measure of how seriously the developers work on the software (Duijnhouwer & Widdows, 2003). One way to check this is to see if there is a separate bug tracker[4] for the software and how actively it is being used by both the developers and the users. When the developers use it but hardly any users seem to participate, the users may not be pointed in the right direction to report problems. Aside from community support, larger or more popular projects may have paid support options. The software is free to use, but the user has the option to get professional support for a fee, either on a

service-agreement basis where a subscription fee is paid for a certain period of time, or a per-incident fee for each time the user calls on support. The project leaders themselves may offer something like this, which is the case for the very popular open source database server MySQL (2005).

There are companies that offer specialised support for certain open source software. This is called third-party support. For example, at the Mozilla support Web page, it can be seen that DecisionOne offers paid support for Mozilla's popular Web browser FireFox, the e-mail client Thunderbird, and the Mozilla Suite (Mozilla, 2005). The fact that paid support exists for an open source product, especially third-party support, is a sign of maturity and a sign the product is taken seriously.

Support for open source software is in most cases handled by the community. The community's support areas are invaluable resources for solving problems (Golden, 2005). Mature products often have paid support options as well if more help or the security of a support contract is required.

Community Support

The usage support is usually found in the community. Things to look for include the following (Golden, 2005):

- Does the program have a separate forum or group for asking installation- and usage-related questions?
- How active is this forum?
- Are developers participating?
- Are questions answered adequately?
- Is there adequate documentation (see the documentation section)?

Responses to questions should be to the point and the responders friendly and helpful. In the process of evaluating software, the evaluator will probably be able to post a question. Try to keep to the etiquette, where the most important rule is to search for a possible answer on the forum before posting a question and to given enough relevant information for others to reproduce the problem (Golden, 2005; Wheeler, n.d.).

The way the community is organised influences the community support's effectiveness. A large project should have multiple areas for each part of the project, but the areas should not be spread to thin. That way, the developers that are responsible for a certain part of the project are able to focus on the relevant area without getting overwhelmed with a large amount of other questions. If the areas are too specialised and little activity takes place in each, not enough people will show interest and questions are more likely to remain unanswered.

Failure support within the project is often handled by a bug tracker by which problems are reported and tracked. Statistical studies have shown that in successful projects, the number of developers that fix bugs in open source software is usually much higher than the number of developers creating new code (Mockus, Rielding, & Herbsleb, 2000).

Paid Support

Paid support might be available from the project team itself (Golden, 2005). There may have been people who have given their opinion about the quality of this support.

One of the strong signs of the maturity of open source software is the availability of third-party support: companies that offer commercial support services for open source products (Duijnhouwer & Widdows, 2003). Some companies offer service contracts, others offer only phone support on a per-incident basis. Check for paid support options whether they will be used or not (Duijnhouwer & Widdows). How the situation may be during actual use of the software is not always clear and it can give a better impression of the maturity of the software.

Documentation

There are two main types of documentation (Erenkratz & Taylor, 2003):

- User documentation
- Developer documentation

User documentation contains all documents that describe how to use the system. For certain applications, there can be different levels in the user documentation, corresponding with different user levels and rights. For example, many applications that have an administrator role have a separate piece of documentation for administrators. Additionally, there can be various user-contributed tutorials and how-tos, be it on the project's Web site or elsewhere. The available documentation should be adequate for your needs. The more complex the software, the more you may need to rely on the user documentation.

The other main type of documentation, which plays a much larger role in open source software than in proprietary applications, is developer documentation. A voluntary decentralised distribution of labour could not work without it (Weber, 2004). The developer documentation concerns separate documents on how to add or change the code, as well as documentation within the source code by way of comments. The comments usually explain what a section of code does, how to use and change it, and why it works like it does. Though this type of documentation may exist for proprietary software, it is usually not public.

If it is possible that you may want to change or add to the source code, this documentation is very valuable. A programmer or at least someone with some experience in programming will be better able to evaluate whether this documentation is set up well, especially by the comments in the source code. It is a good idea to let someone with experience take a look at this documentation (n.d., 2005).

A third type of documentation that is often available for larger server-based applications is maintainer documentation, which includes the install and upgrade instructions. These need to be clear, with the required infrastructure and the steps for installing the software properly explained. This documentation is needed to set up the application. For this type, again, the complexity of the application and its deployment determines the level of documentation that is needed. Documentation is often lagging behind the status of the application since it is often written only after functionality is created, especially user documentation (Scacchi, 2002). It is a good idea to check how often the documentation is updated, and how much the documentation is behind compared to the current status of the software itself.

The documentation for larger projects is often handled by a documentation team. A discussion area may exist about the documentation, giving an indication of the activity level of that team.

Security

Security in software, especially when discussing open source software, has two sides to it. There are people who believe security by obscurity is better, meaning that the inner workings of the software are hidden by keeping it closed source, something that open source obviously does not do. The advocates of security by obscurity see the openness of open source software as a security hazard. Others argue that the openness of open source actually makes it safer because vulnerabilities in the code are found sooner. Open source software gives both attackers and defenders great power over system security (Cowan, 2003; Hoepman & Jacobs, 2005).

Security depends strongly on how much attention the developers give to it. The quality of the code has much to do with it, and that goes for both proprietary and open source software. If the code of proprietary software is not secure, the vulnerabilities may still be found. There are

plenty of examples where this occurs, such as the Microsoft Windows operating system (OS). The vulnerabilities are often found by hackers who try to break the software, sometimes by blunt force or simple trial and error. In this case, a vulnerability might get exploited before the vendor knows about it. The attack is the first clue in that case. The open source software's vulnerabilities, however, could be found by one of the developers or users just by reviewing the code; he or she can report the problem so it can be fixed (Payne, 2002). It is important that the developers take the security of their software seriously and respond swiftly to any reported vulnerabilities.

There are various security advisories to check for bugs in all types of software that make it vulnerable to attacks. A couple of well-known advisories are http://www.securityfocus.com and http://www.secunia.com. Keep in mind that more popular software will have a higher chance of having vulnerability reports, so the mere lack of reports is no proof of its security. On the project's Web site, it can be seen, for instance in the release notes, how serious the project is about security.

Functionality

Though functionality comparison is not specific to open source software evaluation and is properly covered in most traditional software evaluation models, there are some points to take into consideration. Open source software often uses the method described by the phrase "release early and often" (Raymond, 1998). This method enables faster error correction (Weber, 2004) by keeping the software up to date as much as possible. It also encourages people to contribute because they see the result of their work in the next release much sooner (Raymond). However, this often means that the software is incomplete during the first releases, at least more so than is customary with proprietary software. Where vendors of proprietary software will offer full functionality descriptions for their software, open source projects might not have the complete information on the Web site (Golden, 2005). Just like with documentation, the information on the Web site might be lagging behind the actual functionality. Other means of checking the current functionality set might be needed. Fortunately, open source software that is freely available gives the added option of installing the software to enable the full testing of the functionality, an option that is mostly not available with proprietary software, for which at most only limited versions, in terms of functionality or time, are given freely for trying it out.

One problem with open source projects is that the documentation is not always up to date with the latest software. Look beyond the feature list on the Web site to find out what features the software has. Two options are to query the developers and ask the user community (Golden, 2005). Eventually the software itself should be investigated. If it is a Web-based application, an online demo might be available, though installing it on a test environment could be useful because it also gives insight on how well the software installs.

A list of functional requirements for the goal of the software can be used to check if the needed functionality is available. If such a list is not given, there may be one available from technology analyst organisations (Golden, 2005). It is wise to make a distinction in the list between features that are absolutely necessary, where the absence would lead to elimination, and those that would be a plus, which results in a higher score. If there is something missing, there is always the option to build it or have it built.

When comparing functionality, those features that are part of the functional requirements should take priority, but additional features may prove useful later. The features used or requested by the users in the future are not really predictable. While evaluating the software, features may be found in some of the candidates that are very useful for the goal. These can be added to the functional requirements.

Part of the functionality is localisation. The languages to which the interface and documentation are translated are a sign of the global interest taken in the software.

Integration

Duijnhouwer and Widdows (2003) mention three integration criteria. These are most important for software that is being used in collaboration with other software, and for people who are planning on adapting the software to their use, such as adding functionality or customising certain aspects so that it fits better in the organisation's environment. The three criteria are discussed in the next three subsections.

Modularity

Modularity of software means that the software or part of the software is broken into separate pieces, each with its own function. This type of structure has the following advantages:

- Modular software is easier to manage (Garzarelli, 2002; Mockus, Fielding, & Herbsleb, 2002).
- With a base structure that handles the modules well, people can easily add customised functionality without touching the core software.
- Modular software enables the selection of the needed functionality, leaving out those that are not necessary for the intended use. This way, the software can be customised without the need for a programmer.
- Modular software can be used in commercial applications. By making software modular, not everything needs to be given away as open source. It is can be used to give away only parts of software as open source while the add-on modules are sold as proprietary software (Duijnhouwer & Widdows, 2003). This is also called the razor model, as in giving away the razor for free and charging for the blade (Golden, 2005).

Evidence of a modular structure can often be found in several places, such as the source code, the developer documentation, or the download section, where modules might be available for download separate from the core software.

Standards

In the software market, more and more open standards emerge to make cooperation between software easier (Golden, 2005). If the software vendors use these standards in their software, it makes it easier to communicate between different software packages, and to switch between software packages. In some industries, standards are far more important than in others. For some software, there may not even be an applicable standard.

The use of current and open standards in open source software is a sign of the software's maturity (Duijnhouwer & Widdows, 2003). The feature list of the software usually lists what standards are used and with which the software complies.

Collaboration with Other Products

Closely connected to standards is the collaboration with other products. As mentioned before, not every software type has applicable standards, and sometimes the formal standards are not used as much as other formats. Examples of such formats are the Microsoft Word document format, and Adobe's PDF (portable document format). The office suite OpenOffice.org (2005) has built-in compatibility for both formats.

Software Requirements

Most software is written for a specific OS, for example, Microsoft Windows or Linux (Wheeler, n.d.). Certain types of software also rely on other

software, such as a Web server or a database. The requirements of the software will state which software and which versions of that software are compatible. If these requirements are very specific, it could lead to problems if they are incompatible with the organisation's current environment.

THE FUTURE OF OPEN SOURCE SOFTWARE EVALUATION

Open Source Software Evaluation Literature

More is being written on open source software evaluation at the time of writing. For example, another model called the business readiness rating (OpenBRR, 2005), aimed at open source software, was released recently. The research of Crowston and others is still ongoing, so there will be more results in the near future to include in the open source software evaluation process. Given how recent the rest of the literature discussed in this chapter is, it is likely that more will be published on the subject in the next few years.

The Future of Open Source Software

Open source software is being used increasingly by corporations worldwide. There is now some literature available to help with the evaluation of open source software, and the number of articles and models is increasing. With this growth in the field comes more attention from companies, especially on the enterprise level, which will cause more demand for solid evaluation models. Because open source software and the process around it provide much more information than traditional software, there is certainly a need for such models.

This literature will help justify and solidify the position of open source software evaluation in a corporate setting, giving more incentive to use open source software. Most likely, more companies will be investing time and money in its development, like we are seeing today in examples such as Oracle investing in PHP and incorporating this open source Web development language in its products (Oracle, 2005), and Novell's acquisition of SUSE Linux (Novell, 2003). The open source software evaluation literature can help IT managers in adopting open source.

CONCLUSION

The field of open source software evaluation is growing, and with that growth more attention is gained from the large enterprises. With this attention comes more demand for evaluation models that can be performed for these corporations, which will give more growth to the open source software market as well. In this chapter, an overview is given of the current literature and the criteria derived from that literature that can be used in open source software evaluation. For each of the criteria—community, release activity, longevity, license, support, documentation, security, and functionality—this chapter explains why it is important in the market and what to do to evaluate it. This information can be used on its own or in conjunction with more traditional evaluation models and additional information referenced here by companies and individuals that wish to evaluate and select an open source software package. It helps to give insight into the open source software sector.

REFERENCES

Chavan, A. (2005). Seven criteria for evaluating open source content management systems. *Linux Journal.* Retrieved August 9, 2005, from http://www.linuxjournal.com/node/8301/

Cowan, C. (2003). Software security for open source systems. *Security & Privacy Magazine*, *1*(1), 38-45.

Crowston, K., Annabi, H., & Howison, J. (2003). Defining open source software project success. In *Twenty-Fourth International Conference on Information Systems, International Conference on Software Engineering (ICIS 2003)* (pp. 29-33). Retrieved March 30, 2005, from http://opensource.mit.edu/papers/crowstonannabihowison.pdf

Crowston, K., Annabi, H., Howison, J., & Masango, C. (2004). Towards a portfolio of FLOSS project success measures. *Collaboration, Conflict and Control: The Fourth Workshop on Open Source Software Engineering, International Conference on Software Engineering (ICSE 2004)*, 29-33. Retrieved March 30, 2005, from http://opensource.ucc.ie/icse2004/Workshop_on_OSS_Engineering_2004.pdf

Donham, P. (2004). Ten rules for evaluating open source software. *Collaborative Consulting.* Retrieved August 8, 2005, from http://www.collaborative.ws/leadership.php?subsection=27

Duijnhouwer, F., & Widdows, C. (2003). *Capgemini open source maturity model.* Retrieved February 12, 2006, from http://www.seriouslyopen.org/nuke/html/modules/Downloads/osmm/GB_Expert_Letter_Open_Source_Maturity_Model_1.5.3.pdf

Erenkratz, J. R., & Taylor, R. N. (2003). *Supporting distributed and decentralized projects: Drawing lessons from the open source community* (Tech. Rep.). Institute for Software Research. Retrieved August 9, 2005, from http://www.erenkrantz.com/Geeks/Research/Publications/Open-Source-Process-OSIC.pdf

Garzarelli, G. (2002, June 6-8). *The pure convergence of knowledge and rights in economic organization: The case of open source software development.* Paper presented at the DRUID Summer Conference 2002 on Industrial dynamics of the new and old economy—Who embraces whom?, Copenhagen.

Golden, G. (2005). *Succeeding with open source.* Boston: Addison-Wesley Pearson Education.

Hoepman, J., & Jacobs, B. (2005). *Software security through open source* (Tech. Rep.). Institute for Computing and Information Sciences, Radboud University Nijmegen. Retrieved August 9, 2005, from http://www.cs.ru.nl/~jhh/publications/oss-acm.pdf

Mockus, A., Fielding, R. T., & Herbsleb, J. (2000). A case study of open source software development: The Apache Server. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000).* Retrieved on March 30, 2005, from http://opensource.mit.edu/papers/mockusapache.pdf

Mockus, A., Fielding, R. T., & Herbsleb, J. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, *11*(3), 309-346.

Mozilla. (2005). *Mozilla.org support.* Retrieved February 16, 2005, from http://www.mozilla.org/support/

MySQL. (2005). *MySQL support Web site.* Retrieved February 16, 2005, from http://www.mysql.com/support/premier.html

Nijdam, M. (2003). Vijf adviezen voor selectie van oss-compontenten. *Informatie: Maandbladvoor Informatieverwerking*, *45*(7), 28-30.

Novell. (2003). *Novell announces agreement to acquire leading enterprise Linux technology company SUSE LINUX.* Retrieved August 8, 2005, from http://www.novell.com/news/press/archive/2003/11/pr03069.html

OpenBRR. (2005). *Business readiness rating for open source: A proposed open standard to facilitate assessment and adoption of open source software (RFC1).* Retrieved August 10, 2005, from http://www.openbrr.org/docs/BRR_whitepaper_2005RFC1.pdf

OpenOffice.org. (2005). *OpenOffice.org writer product information.* Retrieved August 10, 2005, from http://www.openoffice.org/product/writer.html

Open Source Initiative (OSI). (2005). *Open Source Initiative: Open source licenses.* Retrieved August 9, 2005, from http://opensource.org/licenses/

Oracle. (2005). *Oracle and Zend partner on development and deployment foundation for PHP-based applications.* Retrieved February 12, 2006, from http://www.oracle.com/corporate/press/2005_may/05.16.05_oracle_zend_partner_finalsite.html

Payne, C. (2002). On the security of open source software. *Information Systems Journal*, *12*(1), 61-78.

Raymond, E. S. (1998). The cathedral and the bazaar. *First Monday*, *3*(3). Retrieved March 30, 2005, from http://www.firstmonday.org/issues/issue3_3/raymond/

Scacchi, W. (2002). Understanding the requirements for developing open source software systems. In *IEEE Proceedings: Software*, *149*, 24-29. Retrieved March 30, 2005, from http://www1.ics.uci.edu/wscacchi/Papers/New/Understanding-OS-Requirements.pdf

Van den Berg, K. (2005). *Finding open options: An open source software evaluation model with a case study on course management system.* Unpublished master's thesis, Tilburg University, Tilburg, The Netherlands. Retrieved August 30, 2005, from http://www.karinvandenberg.nl/Thesis.pdf

Weber, S. (2004). *The success of open source.* Cambridge, MA: Harvard University Press.

Wheeler, W. (n.d.). *How to evaluate open source/free software (OSS/FS) programs.* Retrieved February 17, 2005, from http://www.dwheeler.com/oss_fs_eval.html

KEY TERMS

Community: A group of people with shared interests that interact. In case of open source software, the community is the group of developers and users that come together, mostly on a Web site, to discuss, debug, and develop the software.

Documentation: The documents that are associated with a piece of software. There is usually user documentation, in the form of help files, tutorials, and manuals, and there can be developer documentation, such as programming guidelines and documents explaining the structure and workings of the software (source code). In some cases there is administrator documentation, which explains how to install and configure the software. The latter is more important for large pieces of software, where one installation will be used by many users, such as Web applications.

License: An agreement that is attached to the use of a product. In case of software, the software license agreement defines the terms under which you are allowed to use the software. For open source software, there are a number of common licenses, not bound to a specific piece of software, that can be used for almost any type of open source software. These licenses are well known so users and developers usually know the conditions of these licenses.

Maturity Model: Not to be confused with the capability maturity model (CMM), a maturity model as discussed in this chapter is a model that can be used to assess the maturity of a software package, evaluating the software using several criteria.

Software Longevity: The life expectancy of software, measured by various factors among which is its age.

Software Release Activity: The number and significance of releases that are made for a certain software package. A release can be a minor

change such as a bug fix, or a major change such as added functionality.

Software Security: How well a piece of software is built in terms of vulnerabilities and defense against them. Any software will have some type of security hole in it that allows a person, often with hostile intentions, to break into the software and use it for purposes that are unwanted. It is necessary for developers to minimize these holes and fix them if they are discovered. In case of open source software, because the source is public, the users may help in discovery by examining the source code. This, however, also means that a person with hostile intentions can also find these holes by examining the source code. Thus, it is always important to keep a close eye on security.

ENDNOTES

1. http://www.sourceforge.net
2. http://www.freshmeat.net
3. Established using the SourceForge Software Map on April 20, 2005, at http://sourceforge.net/softwaremap/trove_list.php?form_cat=13
4. A bug tracker is an application, often Web based, through which the users can report problems with the software, the developers can assign the bug to someone who will handle it, and the status of the bug can be maintained. Bugzilla is one such package that is often used for this purpose.

This work was previously published in Handbook of Research on Open Source Software: Technological, Economic, and Social Perspectives, edited by K. St.Amant and B. Still, pp. 197-210, copyright 2007 by Information Science Reference (an imprint of IGI Global).

Van den Berg, K. (2009). Open Source Software Evaluation. In S.R. Binford L.R. Binford (Eds.), *Archeology in cultural systems* (pp. 52- 63).