Knowing a great deal is not the same as being smart; intelligence is not information alone but also judgment, the manner in which information is collected and used.

- Carl Sagan
5 - Open Source Software Evaluation:

In this chapter, a model is proposed for evaluating Open Source software using an approach adopted to the unique characteristics of Open Source software. This comparison can be used stand-alone or in conjunction with traditional software evaluation methods.

The criteria that are used were derived from Open Source literature, among which are two Open Source maturity models. In order to understand their importance and value, the section 5.2 gives a description of each criterion. The evaluation process, consisting of the selection of a small number of candidates (the ‘Short list’) and the in depth evaluation of the short listed candidates, is described in sections 5.3 and 5.4.

5.1 - The Criteria:

The criteria for the Open Source software evaluation model were established using Open Source literature on the subject of evaluation of software. Because scientific literature is still somewhat scarce on the subject, web searches were needed to find the required resources. The web searches were done to find the most prominent articles on Open Source maturity models, Open Source success factors and Open Source software evaluation. Two Open Source maturity models were found, as well as three articles giving advice on selecting Open Source software and one research article that investigated Open Source success factors. In order to identify the criteria that give a good general idea of the Open Source software that needs to be evaluated, all criteria were listed and the terms covering the same areas were grouped together. The criteria that were mentioned in some way in at least four out of the six sources were included in this model. The six sources are listed below.

☞ Succeeding with Open Source (Golden, 2005) – This book uses the Navica Open Source Maturity Model.
Towards A Portfolio of FLOSS Project Success Measures (Crowston et al., 2004).
How to evaluate Open Source / Free Software (OSS/FS) Programs (Wheeler, 2005).
Vijf adviezen voor selectie van oss-componenten (Nijdam, 2003).

5.2 - Description of the Criteria:

The criteria are each described in the following subsections, giving background information and explaining why the criterion is important for software evaluation.

5.2.1 – Community:

“One of the most important aspects of open source is the community” (Golden, 2005, p.21). The user community of an open source project consists of the people that use the software and participate in some way. One way of participating is by filing bug reports, which is a report on a problem in the software. This could be something very small, like a typing error in the text, or something large, like a feature that does not work properly. Another way is giving feedback on functionality the user would like to be added. Some users turn into developers over time, and participate in things such as fixing bugs or adding functionality themselves. They cross the line to 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 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 and Widdows, 2003).

5.2.2 - 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:

☞ Their participation in the community
☞ The development itself – writing or changing the source code.

The latter activity is reflected 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 and FreshMeat, where projects can share files with the public, provide information on activity that could be useful to evaluate the release activity (Wheeler, 2005).

An Open Source project often has different types of releases:

☞ **Stable Releases:** the most important type for the end user. This is the version of the software that is 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 not meant for production use because there is a higher possibility of errors.
A project which releases new versions of their 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 roadmap, 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 roadmap 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, 2005).

### 5.2.3 - 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 it usually still full of bugs (Golden, 2005, p.103). The older a project, the less likely the developers will suddenly stop (Duijnhouwer and Widdows, 2003). But age is not always a guarantee of the chance 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 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 doesn’t mean that the activity should ever be slim to none. As mentioned before, no project is ever static (Wheeler, 2005). There’s always something that still needs to be done.
The most important term in this context is ‘copyleft’, introduced by Richard Stallman, where copyright is used to ensure free software and their derivative works remain free (Weber, 2004, p.48). 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 (Weber, 2004, p.48-49). This is also one of the most used licenses. On SourceForge, a large Open Source public repository where over 2,52,000 projects reside, almost 70% uses the GNU GPL as their license.

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 was less restrictive in its ‘copyleft’ statements, allowing libraries to be used in other applications without the need to distribute the source code (Weber, 2004, p. 183).

A ‘non-copyleft’ license that is much heard of is the BSD 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 users 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 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.
5.2.5 – Support:

There are two types of support for a software product:

☞ **Usage Support**: the answering of questions on the 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, p.124).

The way support is handled is a measure of how seriously the developers work on the software (Duijnhouwer and Widdows, 2003). One way to check this is to see if there is a separate bug tracker 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 (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 a Open Source product, especially third party support, is a sign of maturity and a sign the product is taken seriously.
5.2.6 – 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 which 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 and 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. 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, and 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.

5.2.7 – 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 ‘Release Early and Often’ (Raymond, 1998b). This method enables faster error correction (Weber, 2004, p.80), 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, 1998b). 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 website (Golden, 2005, p100-101). Just like with documentation, the information on the website 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 which enables the full testing of the functionality, an option that is mostly not available with proprietary software, where at most only limited versions, in terms of functionality or time, are given freely for trying out the software.

5.2.8 – 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 those 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.

MODULARITY:

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

- Modular software is easier to manage (Mockus et al., 2002; Garzarelli, 2002), and with a base structure that handles the modules well, people can easily add customised functionality without touching the core software. This way the software can still be upgraded without losing any custom code, because the module works alongside the core software which need not be changed. It also encourages contribution to the software. When someone writes a new module for his own use he may contribute this module back to the community later so others can use it too.
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.

Use in commercial software: 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 and Widdows, 2003), also called the ‘razor’ model giving away the razor for free and charging for the blade (Golden, 2005, p.34).

**STANDARDS:**
In the software market more and more open standards emerge to make cooperation between software easier (Golden, 2005, p.190). 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 and Widdows, 2003).

**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. For example, the Microsoft Word document format, the .doc, is probably the most used for exchanging editable documents. Software such as OpenOffice has the ability to import and export to the .doc format, enabling its users to exchange documents with Microsoft Word users (Varian and Varian, 2003; OpenOffice.org, 2005).

**SOFTWARE REQUIREMENTS:**
Most software is written for a specific Operating System (OS), for example Microsoft Windows or Linux (Wheeler, 2005). 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.

5.3 – Selection:

When evaluating software, a small list of softwares is needed which is evaluated fully. In order to get this list, a selection is performed on the list of all possible softwares to get the ‘short list’ (Golden, 2005, p.96-87). There are five sources that can be used to populate a software list (Golden, 2005, p.94-96):

- Search Open Source Project Portals – for example SourceForge and FreshMeat.
- Search the Web – this can also give pages about projects and user opinions.
- Ask Open Source Developers.
- Post to Mailing Lists.
- Ask Vendors.
<table>
<thead>
<tr>
<th><strong>Criterion</strong></th>
<th><strong>Background</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Community</strong></td>
<td>The driving force and main resource of an Open Source project.</td>
</tr>
<tr>
<td><strong>Release activity</strong></td>
<td>Shows development activity and progress.</td>
</tr>
<tr>
<td><strong>Longevity</strong></td>
<td>Indication of stability and chance of survival.</td>
</tr>
<tr>
<td><strong>License</strong></td>
<td>GNU GPL is a well known public copyleft license. Copyleft ensures code and derivatives stay under that license.</td>
</tr>
<tr>
<td><strong>Support</strong></td>
<td>Community and paid support, answering questions and failure support.</td>
</tr>
<tr>
<td><strong>Documentation</strong></td>
<td>User manuals and tutorials – developer documentation about code structure and coding guidelines.</td>
</tr>
<tr>
<td><strong>Security</strong></td>
<td>Openness vs. obscurity. Security needs to be taken seriously.</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>Release early and often can lead to incomplete products. Feature information not always clear.</td>
</tr>
<tr>
<td><strong>Integration</strong></td>
<td>Modularity, standards and collaboration with other products.</td>
</tr>
<tr>
<td><strong>Goal and origin</strong></td>
<td>Does the project team’s goal fit with the intended use? Gives an indication of how serious the developers are about the project.</td>
</tr>
</tbody>
</table>