UPGRADE
The European Journal for the Informatics Professional
http://www.upgrade-cepis.org
Vol. VI, issue No. 3, June 2005

UPGRADE is the European Journal for the Informatics Professional, published bimonthly at <http://www.upgrade-cepis.org/>

UPGRADE is the anchor point for **UP**ENET (**UP**GRADE European NETwork), the network of CEPIS member societies' publications, that currently includes the following ones:
• **Mondo Digitale**, digital journal from the Italian CEPIS society AICA
• **Novática**, journal from the Spanish CEPIS society ATI
• **OCG Journal**, journal from the Austrian CEPIS society OCG
• **Pliroforiki**, journal from the Cyprus CEPIS society CCS
• **Pro Dialog**, journal from the Polish CEPIS society PTI-PIPS

Publisher
UPGRADE is published on behalf of CEPIS (Council of European Professional Informatics Societies, <http://www.cepis.org/>) by **Novática** <http://www.ati.es/novatica/>, journal of the Spanish CEPIS society ATI (*Asociación de Técnicos de Informática*, <http://www.ati.es/>)

UPGRADE monographs are also published in Spanish (full version printed; summary, abstracts and some articles online) by **Novática**, and in Italian (summary, abstracts and some articles online) by the Italian CEPIS society ALSI (*Associazione nazionale Laureati in Scienze dell'informazione e Informatica*, <http://www.alsi.it>) and the Italian IT portal **Tecnoteca** <http://www.tecnoteca.it/>

UPGRADE was created in October 2000 by CEPIS and was first published by **Novática** and **INFORMATIK/INFORMATIQUE**, bimonthly journal of SVI/FSI (Swiss Federation of Professional Informatics Societies, <http://www.svifsi.ch/>)

ISSN 1684-5285

Monograph of next issue (August 2005):
"Normalisation & Standardisation in IT Security"
(The full schedule of **UP**GRADE is available at our website)

Monograph: *Libre* Software as A Field of Study
(published jointly with Novática*, in cooperation with the European project CALIBRE)
Guest Editors: *Jesús M. González-Barahona and Stefan Koch*

MOSAIC

UPENET (**UP**GRADE European NETwork)

* This monograph will be also published in Spanish (full version printed; summary, abstracts, and some articles online) by **Novática**, journal of the Spanish CEPIS society ATI (*Asociación de Técnicos de Informática*) at <http://www.ati.es/novatica/>, and in Italian (online edition only, containing summary, abstracts, and some articles) by the Italian CEPIS society ALSI (*Associazione nazionale Laureati in Scienze dell'informazione e Informatica*) and the Italian IT portal Tecnoteca at <http://www.tecnoteca.it>.

The Challenges of Using Open Source Software as A Reuse Strategy

Christian Neumann and Christoph Breidert

This paper compares the benefits of adapting open source software to internal and commercial reuse strategies. We propose a course of action that can be used for technical and economical evaluation. The advantages, disadvantages, and risks of these basic strategies are investigated and compared.

Keywords: Commercial-Off-The-Shelf, COTS, FOSS, Free and Open Source Software, Software Engineering, Software Reuse.

1 Introduction

Reusing existing software is an important part of modern software engineering, promising cost reduction, faster time to market, and improved quality. One study shows that productivity can be increased by up to 113%, the average fault rate can be lowered by 33% and time to market can be shortened by 25% [11]. Another study reports a 51% defect reduction, a 57% productivity increase, and a 57% faster time to market [16].

The decision whether or not to introduce reuse into the engineering process depends on two important conditions: components must meet certain technical requirements and their usage must be economically viable. This paper describes a technical course of action for choosing the best alternative out of three rudimentary reuse strategies: the development and usage of in-house components, the adoption of external commercial components (the so Called Commercial-Off-The-Shelf, COTS, components), and the integration of Free and Open Source Software, FOSS).

Recent studies have discussed the use of FOSS and its effect on total cost of ownership and quality. But these studies mainly focus on the use of operating systems such as Linux, or application software like the Apache web server. For example, one study estimates that using FOSS can lead to savings of about 12 million Eur (one fifth of the cost of using commercial applications) over a five-year period [7]. Another study points to savings of about 33% compared to commercial software over a two-year period [19].

However, with the exception of these studies, there is hardly any knowledge about the integration of FOSS in the software engineering process. The free concept of FOSS opens the door to new strategies and business models that can be executed at almost every phase of the development process, such as using FOSS as a basis for future product line development or contributing software components to the FOSS community. The last point is especially interesting for companies because it enables the maintenance process of software artefacts to be outsourced.

This paper focuses on two different business models: system vendors that develop specific software to customer order, and software companies which develop standard software.

2 Adapting Reuse Strategies

Figure 1 illustrates the course of action to be taken before a decision is made. The decision making process is divided into three phases: sighting, adaptation, and comparison. In the first two phases, which focus on technical aspects, the decision is taken as to whether a component provides the desired functionality or, if not, whether it can be extended. In phase three the economic aspects of the remaining candidates are investigated thoroughly and the implications for different business models are discussed.

2.1 Technical Analysis

First of all the technical requirements of the project need to be specified. This specification is largely deduced from the desired functionality and contains the architecture, I/O functionality, and business logic. Based on this specification, the software developer identifies possible places where components can be hooked into the architecture of the application. A good place to start searching is in I/O procedures because almost every program uses file operations,

Christian Neumann is a PhD student at the Dept. of Information Business at the Vienna University of Economics and Business Administration, Austria. He received his masters degree in Engineering and Management from the University of Karlsruhe, Germany. His research interests include quality of open source projects, usability of frameworks, cost estimation, and software investment analysis. He worked for several years as a software engineer for a major German IT company. <christian.neumann@ai.wu-wien.ac.at>.

Christoph Breidert holds a PhD from the Dept. of Information Business at the Vienna University of Economics and Business Administration, Austria. He received his masters degree in Engineering and Management from the University of Karlsruhe, Germany. He has several years experience in software development of large-scale J2EE projects.<christoph.breidert@ai.wuwien. ac.at>.

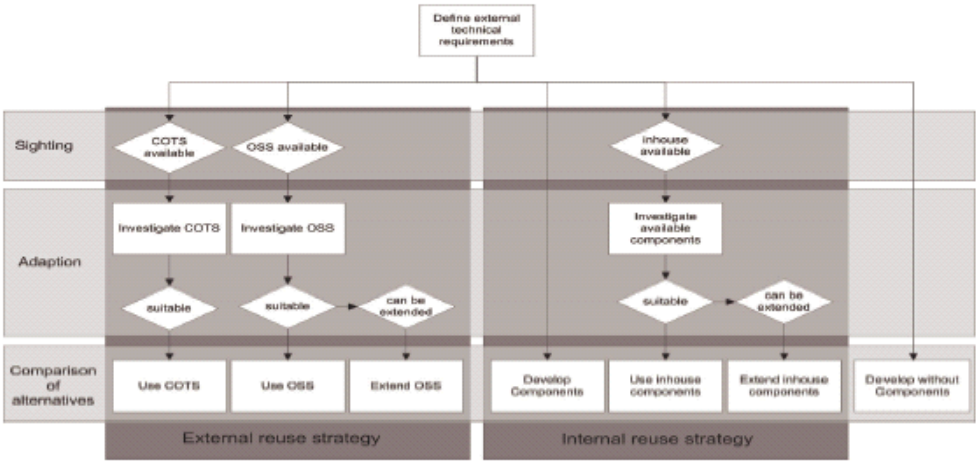

Figure 1: Model for Evaluating Reusable Components.

database access, or a Graphical User Interface (GUI).

Phase one of the analysis consists of a sighting in which possible reusable components are identified. In the case of internal components, sources of information for this task are reuse repositories or implicit knowledge of former projects. The search for external components can be performed by using the Internet, asking newsgroups, visiting exhibitions, or reading magazines. A good place to start the search for FOSS are some of the great online repositories such as sourceforge, <http://www.sourceforge.net>, or Apache, <http://www.apache.org>. At these repositories FOSS can be searched for by theme, programming language, or keywords. The Apache Foundation in particular hosts a wide range of ready to use components, some of which have evolved into de facto standards (log4j, xerces, xalan, etc.).

The second phase takes a deeper look into the functionality provided by the components that were identified in phase one. Possible sources of information are documentations, specifications, and examples. If the provided functionality does not comply with the technical requirements we need to check whether their components can be extended. Naturally this is only applicable for FOSS and internal components, because COTS do not provide the source code required for modification. Using the components unchanged is called black-box reuse; modifying the components before use is called white-box reuse [22].

In the case of FOSS it is very important to address the following issues [28]: future functional upgradability, open-standard compatibility, adaptability and extensibility, and reliability. To address these issues it is necessary to take a deeper look into design and architecture which we would hope to be described in the user manual. Unfortunately, many FOSS projects suffer from bad documentation which makes it very difficult to learn or extend their components, making their integration, and in particular their extension or adaptation, difficult. Therefore the existence of good documentation, examples, articles or newsgroups is indispensable for a fast and cost effective integration. There are several systematic ways of searching for FOSS components (e.g. <http://www.amosproject.org>) and evaluating FOSS [9].

To ensure the quality of reusable software, especially in long-term projects such as frameworks in product lines, the source code must be inspected. This can be done by using quality indicators such as object-oriented metrics or commentaries [4][5][10][18].

All possible candidates that have passed the first two stages are compared in phase three. These two phases may require a great deal of effort, but the information gathered provides the basis for our final economic and managerial analysis.

2.2 Specific Software

Imagine a system vendor that implements a highly customised solution for a client; for example, a monitoring system for a diversified IT (Information Technology) landscape. The requirements specification forms the basis for the offer and every additional functionality of the customer leads to a change request that has to be renegotiated every time.

The reduction of development effort and the resulting decrease of costs is the most important managerial aspect to be considered. Time to market may be another constraint that needs to be kept in mind. The use of existing components, both internal and external, is therefore extremely advisable because it is cheaper to reuse them than to implement the same functionality over and over again, and the development schedule can also be shortened. Developing internal reusable components is not cost-efficient because their intended functionality is tailored to just one specific case and it is highly unlikely that they will be reusable in future projects.

The integration of COTS or FOSS also enables systems with highly sophisticated technologies to be produced even when no in-house knowledge is available [2]. Security mechanisms in particular have proven so difficult to implement correctly that developers should be spared the risk of producing major bugs by using suitable COTS [17]. However, the integration of COTS invariably involves license costs.

FOSS provides the same advantages as COTS but there is no license to pay for. The impact of a FOSS component license is not significant in the case of specific software.

Even if the application is an extended version of a FOSS component covered by a copyleft license, it is of no real significance. A copyleft license does not prevent the vendor from selling a derived product and the purchaser has the right to redistribute the program and obtain its source code. In the case of developing specific applications it is common practice to give the source code to the customer anyway. And the customer has nothing to gain from giving its customised solution away to anyone else. But we will later be pointing out that while FOSS licenses are not significant in the case of specific software, they do have implications on the development of proprietary software.

To sum up, in the case of specific software the overall costs for the different types of components must be estimated and the cheapest alternative should be chosen.

2.3 Proprietary Software

Now imagine a software company that has built up extensive knowledge in a specific field – e.g. document management – and wants to provide a desktop application that will help to organise everyday paperwork. The company's marketing division has spotted additional demands, – e.g. other office work tasks – which should be integrated into the application. The company wants to extend its market share and sell licenses that are valid for one workplace only. This situation is completely different from developing specific software because the time horizon is much longer and the business model is based on selling licenses.

The tasks for the software engineer are the same as before: To identify the required functionalities, spot possible components, and investigate the candidates. But in addition it is essential to identify and specify future requirements very thoroughly, especially if the components are an important part of the design, such as a framework architecture for a product line development. To be open for upcoming features the components must be highly customisable, modular and trustworthy.

Developing internal components for reuse is the most flexible reuse alternative with regard to reusability and functionality. As components can be designed from scratch, their functionality is tailored to suit the company's needs. The greatest disadvantage of establishing an internal reuse strategy is the very high initial investment for developing reusable components. Due to the great need for modularity, software quality, and documentation, the cost of developing reusable software can be significantly higher than if the same functionality is developed for a single application only. One study indicates that the break-even point can be reached after just three times of reusing software components [1]. For a successful reuse strategy a repository for the reusable components is mandatory and this must be integrated in the development process. This can lead to higher organisational costs.

Choosing COTS does not require any initial investment apart from purchasing the license. But there are several disadvantages outweighing this advantage.

Flexibility is limited as components cannot be adapted to meet special needs. Additional functionality must be realized in wrappers. Development of this glue code can take up to three times as long as the in-house development of the software [24][27].

The fact that COTS are black-boxes gives rise to two important risks: the risk of undetermined interior behaviour and the risk of interaction with the environment. The first may lead to invalid/wrong results or even to unreliability of the whole application. To prevent this from happening, a great many tests need to be conducted, leading to an increase in integration costs. The latter risk gives rise to a number of security risks such as accessing unauthorised resources, accessing resources in an unauthorised manner, or abusing authorised privileges [30][17].

Another major problem is maintenance since COTS consumers depend on the vendors' efforts in this regard [2]. The frequency and number of updates can be uncertain, making it very difficult for the consumer to maintain the application. Therefore it is very important to have an overview of the development and maintaining cycles of COTS so that they can be synchronised with the actual development process. For these reasons the additional costs involved in maintaining COTS-based systems can equal or even exceed those of custom software development [24].

Adapting existing FOSS components can be the basis for a company's reuse strategy because FOSS combines the advantages of an internal reuse and COTS strategy: No initial investment is needed, existing functionalities can be used, and the source code is available.

The free availability of source code is what makes this strategy so powerful. On the one hand it provides the necessary flexibility to extend or adapt FOSS components and on the other hand it enables inspection and debugging which reduces the risk of unexpected behaviour. Despite the cost of learning and adaptation there is no initial investment needed for integrating the components into the engineering process. Furthermore, the company can cut its maintenance costs because maintenance is carried out by the community. But these steps need to be given careful consideration because any know-how contained in the software may become generally available to the community and other users (see below).

The main drawback of COTS, the unspecified behaviour, also applies for FOSS. But as the source code is available, bugs and flaws can be fixed immediately.

And even if the community no longer supports the product, users can extend and maintain it on their own.

Other advantages stem from the FOSS developing process, which is characterised by the peer-review principal [23]. In a peer-review approach the program is reviewed by at least two experts (in a vivid FOSS community probably more). This critical judgement helps to improve the design, functionality and quality of the program. Furthermore, openness of source code is necessary for security applications because the user can verify the functionality and reliability of the program, such as encrypted data transmission.

But availability of source code does not necessarily mean good programs. A program is only as good as its programmers. It is therefore necessary to take a close look at the

community's activities in terms of the number of contributors involved, the project's maturity (alpha, beta, stable), and the frequency of releases [14][15].

Only projects with several contributors, a high degree of programming activity, and well defined road maps are worth using in company's product lines [21]. Projects that only have a handful of contributors or where the contributed work is highly concentrated may die or remain in an unstable state.

In this case any benefit of using FOSS is lost as there is no community to maintain and evolve the product. The integration of early releases (snapshot release, beta, etc.) should be avoided as these components may be buggy, or future changes in design or functionality may require an additional integration effort.

Other important aspects to be considered are legal issues. Usage in commercial applications gives rise to a number of issues that need to be addressed. Richard Stallman was the first person to coin the term "free software" [26]. In his view, FOSS software, and especially the programs that adapt it, should be distributed freely. This idea leads to the well known GNU General Public License (GPL) which means that the any program using GPL software is covered under the same license. As we mentioned above, selling a GPL covered application is not prohibited but the company does not hold a copyright preventing the user from redistributing the software.

Furthermore the source code of the derived work must be available to anyone using this software. This can be a no-go criteria for the usage of GPL products in a proprietary application. The Library/Lesser GNU Public License (LGPL) is less restrictive and allows closed software using LGPL products to be distributed, but any changes made to the FOSS must be contributed to the community. The least restrictive one is the Berkeley-license, BSD (and its derivatives), which allows the FOSS to be used and modified without having to publish the source code of the modification [6][9][8].

In the case of the document management application we mentioned above, using a copyleft license could lead to a decrease in market share, because the software can be distributed freely and, even worse, the knowledge contained in the documents managed must be made available to everyone using the program. The matter of whether a program that uses GPL components is a derived work or not is a thorny one and currently the subject of much debate [29][12][25]. The use of GPL should therefore be avoided. The LGPL is less restrictive but the constraint of publishing changes to the FOSS community must be carefully considered. Even small changes may reveal insight into the company's core business. So the adaptation of BSD style FOSS should be preferred as this is the most flexible license.

3 Conclusion and Further Work

We have looked into and compared three different reuse strategies and identified the overall costs for adapting reusable components as a primary issue, especially for the development of highly customised specific software. In the case of proprietary software, additional aspects need to be considered. The integration into a product line requires very flexible, modular, and reliable components. Therefore COTS is not suitable due to the drawbacks of black-box reuse. Using FOSS as white-box reuse is a good alternative to an internal reuse strategy because it promises savings both for maintenance and for the implementation of upcoming features. But the impact of the various FOSS licenses must be given careful consideration.

Unfortunately we are unaware of any empirical studies that have compared the overall costs of the three reuse strategies. The reason for this lack of knowledge is obvious. The functionality of the applications investigated ideally need to be identical and the development effort needs to be measured. It is very difficult to find enough comparable projects and companies that are willing to publish internal metrics that can be used to determine their productivity or development methods.

Another way of gathering information is by means of an experiment. An experiment can be used to compare at least two different settings supervised by an independent scientist. The nature of software reuse involves long-term activity which makes it very difficult to conduct such a test in an academic setting (three groups of students developing the same application using different reuse strategies). Furthermore, any such experiment would produce an indicator rather than a substantiated result since the universe is too small.

We suggest comparing the different reuse strategies with existing models for effort estimation, e.g. the COCOMO II (COnstructive COst MOdel) model [3]. This model is based on the evaluation of over hundred software projects and contains a module for software reuse. The impact of several quality indicators – e.g. documentation and understandability – can be integrated. There is some research currently in progress into the economic evaluation of FOSS using estimation models [13][20].

References

[1] T. J. Biggerstaff. "Is technology a second order term in reuse's success equation?" In Proceedings of Third International Conference on Software Reuse, 1994.

[2] Barry Boehm and Chris Abts. "COTS integration: Plug and pray". IEEE Computer, 32(1):135–138, 1999.

[3] Barry W. Boehm, Chris Abts, A. Windsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald Reifer, and Bert Steece. "Software Cost Estimation with CoCoMo II". Prentice Hall PTR, Upper Saddle River, 1 edition, 2000.

[4] Samuel Daniel Conte, H.E. Dunsmore, and V.Y. Shen. "Software Engineering Metrics and Models". The Benjamin/Cummings Publishing Company, Menlo Park, CA, 1986.

[5] Norman E. Fenton. "Software Metrics - A Rigorous Approach". Chapman & Hall, London, 1991.

[6] Martin Fink. "The Business and Economics of Linux and Open Source". Prentice Hall, Upper Saddle River, 2002.

[7] Brian Fitzgerald and Tony Kenny. "Developing an information systems infrastructure with open source". IEEE Software, 21(1):50–55, 2004.

[8] Christina Gacek and Budi Arief. "The many meanings of open

source". IEEE Software, 21(1):34–40, 2004.
[9] Bernard Golden. "Succeeding with Open Source". Addison-Wesley, Boston, 2005.
[10] B. Henderson-Seller. "Object-Oriented Metrics: Measures of Complexity". Prentice Hall, Upper Saddle River, NJ, 1996.
[11] Emmanuel Henry and Benoit Faller. "Large-scale industrial reuse to reduce cost and cycle time". IEEE Software, 12(5):47–53, 1995.
[12] Till Jaeger and Carsten Schulz. "Gutachten zu ausgewählten rechtlichen aspekten der open source software". Technical report, JBB, 2005. <http://opensource.c-lab.de/files/portaldownload/Rechtsgutachten-NOW.pdf>.
[13] S. Koch and C. Neumann. "Evaluierung und aufwandsschätzung bei der integration von open source software-komponenten". In Informatik 2005 -Beiträge der 35. Jahrestagung der Gesellschaft für Informatik e.V. (GI), Lecture Notes in Informatics (LNI) , Gesellschaft für Informatik (GI), 2005. (To appear.)
[14] Stefan Koch. "Profiling an open source project ecology and its programmers". Electronic Markets, 14(2):77–88, 2004.
[15] Stefan Koch and Georg Schneider. "Effort, cooperation and coordination in an open source software project: Gnome". Information Systems Journal, 12(1):27–42, 2002.
[16] Wayne C. Lim. "Effects of reuse on quality, productivity, and economics". IEEE Software, 11(5):23–30, September 1994.
[17] Ulf Lindvist and Erland Jonsson. "A map of security risks associated with using COTS". IEEE Computer, 31(6):60–66, June 1998.
[18] M. Lorenz and J. Kidd. "Object Oriented Metrics". Prentice Hall, Upper Saddle River, N.J., 1995.
[19] T.R. Madanmohan and Rahul De. "Open source reuse in commercial firms". IEEE Software, 21(1):62–69, 2004.
[20] C. Neumann. "Bewertung von open source frameworks als ansatz zur wiederverwendung". In Informatik 2005 - Beiträge der 35. Jahrestagung der Gesellschaft für Informatik e.V. (GI), Lecture Notes in Informatics (LNI) , Gesellschaft für Informatik (GI), 2005. (To appear.)
[21] Jeffrey S. Norris. "Mission-critical development with open source software: Lessons learned". IEEE Software, 21(1):42–49, 2004.
[22] Ruben Prieto-Diaz. "Status report: software reusability". IEEE Software, 10(3):61–66, May 1993.
[23] Eric S. Raymond. "The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary". O'Reilly and Associates, Sebastopol, California, 1999.
[24] Donald J. Reifer, Victor R. Basili, Barry W. Boehm, and Betsy Clark. "Eight lessons learned during COTS-based systems maintenance". IEEE Software, 20(5):94–96, 2003.
[25] Gerald Spindler and Christian Arlt. "Rechtsfragen bei Open Source". Schmidt, Köln, 2004.
[26] Richard Stallman. "Free Software, Free Society: selected essays of Richard M. Stallman". GNU Press, Boston, 2002.
[27] Jeffrey M. Voas. "The challenge of using cots software in component based development". IEEE Computer, 31(6):44–45, June 1998.
[28]Huaiqing Wang and Chen Wang. "Open source software adoption: A status report". IEEE Software, 18(2):90–95, March/April 2001.
[29] Ulrich Wuermerling and Thies Deike. "Open source software: Eine juristische risikoanalyse". Computer und Recht, (2):87–92, 2003.
[30] Qun Zhong and Nigel Edwards. "Security control for COTS components". IEEE Computer, 31(6):67–73, June 1998.

