Identifying and Improving Reusability Based on Coupling Patterns

Andrea Capiluppi and Cornelia Boldyreff

Centre of Research on Open Source Software – CROSS
Department of Computing and Informatics
University of Lincoln
{acapiluppi,cboldyreff}@lincoln.ac.uk

Abstract. Open Source Software (OSS) communities have not yet taken full advantage of reuse mechanisms. Typically many OSS projects which share the same application domain and topic, duplicate effort and code, without fully leveraging the vast amounts of available code.
This study proposes the empirical evaluation of source code folders of OSS projects in order to determine their actual *internal* reuse and their potential as shareable, fine-grained and *externally* reusable software components by future projects.
This paper empirically analyses four OSS systems, identifies which components (in the form of folders) are currently being reused internally and studies their coupling characteristics. Stable components (i.e., those which act as service providers rather than service consumers) are shown to be more likely to be reusable. As a means of supporting replication of these successful instances of OSS reuse, source folders with similar patterns are extracted from the studied systems, and identified as externally reusable components. The intended users are members of the OSS development community.
Based on the empirical study of the OSS systems and observations made during the study, four practical courses of action are recommended in order to enhance the reusability of current folders that have not been identified as potentially reusable, both from an internal and external standpoint.

1 Introduction

Reuse of software components is one of the biggest promises of software engineering [3]. Enhanced productivity, increased quality and improved business performance are often pinpointed as the main benefits of developing software from a stock of reusable building blocks [33].

Recently, practical approaches to commercial software reuse have included both in-house and COTS-based approaches. Many companies have already successfully produced and reused in-house components, in the forms of documentation, system and components design, source code, and so on, which are kept as valuable assets and not made available elsewhere [25, 29]. One of the most critical aspects of successful in-house reuse is the long-term commitment of the management [30].

Reuse of small scale compoments, e.g. functions of programming language libraries, either in-house or externally produced has been common practice since high level languages have been in use. A new possibility for reusability of external components has

arisen through the exploitation of both COTS and OSS in what can termed "whole system reuse". New products can be developed based on existing systems, either on a closed-source basis (e.g., commercially available COTS [31]), or by reusing entire OSS systems, such as web-server Apache, the MySQL database management system, or the PHP language [37]. In the latter case, OSS systems also provide the source code underlying the system, and the code can be modified before reuse. One of the drawbacks of this approach is that entire systems are reused, even though only a subset of their functionalities may be required in the new system. Commercial companies must address these issues in order to take advantage of the proprietary-COTS shift: *whether to use* COTS, *how to use* COTS, and *what to modify* in their in-house systems to cope with COTS [10] and the same questions apply in the selection of OSS component reuse.

Although much attention has already been focused on the study of reusable COTS, including OSS components in corporate software production, the reusability of software "from" OSS projects "in" other OSS projects has only started to draw the attention of researchers and practitioners in OSS communities [20]. While a huge amount of code is daily created, modified and stored in OSS repositories, software reuse is rarely perceived by OSS developers as a critical factor in their projects, nor is the source code of other projects considered as a potential way to build new OSS systems upon existing ones. For different and composite reasons [34], briefly recounted here in the following Sections, several OSS projects typically address the same software need independently. For example, a search for the "*email client*" topic on the SourceForge site will result in more than 500 different projects being listed, each implementing some features of the same topic. Duplication of coding effort therefore is currently producing similar products with little sharing of the basic building blocks or the larger subsystems. In order to address this missing "reuse" link in OSS projects, the objective of this paper is to provide OSS communities with a technique for identifying and benefiting from reusable components (under a "design with reuse" perspective [26]).

2 Definitions and Approach

The terminology and definitions used in this paper are extracted from similar studies in the literature, for example, the definition of *coupling* (intended for both object-oriented [1, 21] and procedural [13] languages) and the notion of *instability* of source packages [17]. In this section an overview of these terms is given, as they will be used throughout this work.

- *Source function*: basic unit of source code; this term is used to refer to procedures, subroutines, but also OO methods.
- *Source file*: any file with at least one source function.
- *Source folder*: any folder containing at least one source file [9]. The term *module* is used to refer to source code functions, files and folders.
- *Folder structure*: from the perspective of file naming, code organisation and storage, this is the tree structure composed of elementary components (source files, source folders). The root of the tree is represented by the parent folder [9, 8].
- *Extensibility of a source folder*: following Martin[24] who defines *extensibility* as the number of concrete and abstract classes in a package, we define the number of

source files contained in a folder as the *extensibility* of that folder. This attribute serves to characterise the potential usefulness of a source folder. Ideally, one would want to reuse folders with large extensibility, i.e., with a large number of similarly scoped functionalities (in the forms of files or functions), rather than a number of related smaller folders.

– *Coupling*: this is a measure of interconnection among modules in a software structure [33]. In this study, three types of coupling are extracted, based on the definitions of common coupling [36]:
  i. The **dependency relationship** is based on source files, and describes, for each file, how many and which other files are currently including it.
  ii. The **include relationship** is also based on source files, and describes the number of external files that a specific source file includes in its declarations.
  iii. The **function call relationship** is based on source functions, and describes the relationship among functions or procedures. It produces as a result the representation of calls within functions. These couplings were extracted via the Doxygen engine, and, albeit related, they represent different metrics for the links among modules.
– The number of *Afferent Couplings (Ca*, or in-bound coupling) of a source folder represents the sum of other source folders that depend on it, and it is an indicator of the its *responsibility*. [17] used this metrics for OO languages and specifically for packages. In the following, the focus is on source folders as packages of a system, even if the system is written in procedural languages.
– The number of *Efferent Couplings (Ce*, or out-bound coupling) of a module represents how many other modules it depends on, and it is an indicator of the folder's *independence* [17].
– The *Instability (I)* of a module is the ratio of efferent coupling (Ce) to total coupling (Ce + Ca) such that $I = \frac{Ce}{(Ca + Ce)}$. This metric is an indicator of the folder's resilience to change [17]. The range for this metric is 0 to 1, with $I = 0$ indicating the lowest instability for a folder and $I = 1$ indicating a completely unstable folder [17]. Since Ca and Ce are measured at the folder level, and couplings among folders may greatly vary due to larger or smaller amount of calls, *weighted* instability factors will be introduced below, termed wCa and wCe.

3 Case Studies – Evolutionary Analysis

The first part of this study has been performed over all the public releases of four large OSS projects, and is specifically targeted at understanding the structural relationships among source folders. It can be observed that both MPlayer and XMMS share the same functionalities, yet they are developed by independent teams of developers. This is not an isolated case in the OSS environment; there are several small projects (such as kftp, gftp, sftp, among others, which are all dealing with the *File Transfer Protocol* management) and large projects (such as the desktop GUIs, KDE and GNOME) which are being developed in parallel, without sharing or reusing code of other similar-scoped projects. The reasons discovered by past works [34] are enough for developers to start their own project and duplicate efforts. This could mean also that the reuse of code

written by others has to overcome similar obstacles, apart from the technical ones, as already evaluated by [20].

In terms of activity and code released, it was observed that the Arla project spans some 8 years of development, Gaim approximately 6.5 years, MPlayer 4.5, while XMMS 5 years. In terms of productivity, a high frequency of releases was visible for the Gaim and MPlayer projects (on average, more than one release per month), while it is lower in the case of Arla (less than one per month). The XMMS case finally shows that a new release has been available on average every two months. This general productivity trend has had a repercussion on size achieved (in LOCs) and overall number of source folders found in the latest observed release.

In terms of developers, it was observed that the MPlayer project was the most successful in forming an OSS community providing code patches, new functionalities and bug fixes (210 developers). A direct link between the community formed and the size achieved was also detected in the overall size at the latest observed release: in the cases of the 4 projects studied here, larger communities usually achieve larger systems, apart from the Gaim case (25 developers, 235 kLOCs), where a smaller community has achieved a larger system than those developed by other, larger communities (Arla – with 83 developers, 215 kLOCs – and XMMS – 43 developers, 110 kLOCs).

The last row of table 1 shows which folders are already successfully reused across the selected OSS systems. The most notably folders are the following:

1. *libraries of the C language (libc)*: they provide generic functionalities, like the I/O output (the module “stdio.h”), or the stub functions for socket communication (contained in “socket.h”). In this work, all the connections involving calls to elements of the generic libc libraries are, for simplicity, redirected to a generic “libc” folder;
2. *localisation/international folder*: the code contained in this specific subsystem translates the messages, or the interfaces, of the application in the local language of the user. OSS projects using code of this subsystem typically include it in a folder named “intl”.

3.1 Source Folders as Reusable Units

Empirical findings reported in [23] demonstrate that object-oriented packages show four basic patterns (“*pure client*”, “*pure server*”, “*hybrid*” and “*silent*”), based on whether they mostly require, or are called by, other packages. Albeit the cited work deals with Java packages and creates a taxonomy of components, the present work expands these findings in two ways:

1. it considers the folders of procedural languages as modules: when asked, the developers of the XMMS case study confirmed that source folders serve to them as place-holders for “similar-scoped” source files[1]. The “wav” folder, for instance, keeps all the source files for the wav audio file format.
2. it evaluates coupling among folders to build an instability index: folders with lowest instability index are identified as candidates for reuse.

[1] Reported from conversations, email correspondence and private communication

In all the case studies apart Arla (see below), an initial value of at least 80% of all the couplings are contained inside the same source folder. Considering only the "function calls" coupling, this value is at least 90%. On a parallel level, all the analysed systems show an initial pattern of growth in number of source folders, and a decreasing coupling pattern: the overall amount of intra-folder couplings decreases while the system increases in size. This recalls the results of architectural erosion mentioned in [27]; as systems depart from the "*initial architecture's intent and conceptual integrity*", couplings connect many other folders, and the whole architecture becomes much harder to understand and maintain.

The Arla case is an outlier, and shows a complex and intertwined system already from its initial releases, where half of the couplings affect two or more source folders. All the other factors being equal, this system is going to experience less externally reusable folders, since most of the existing folders are already linked into a complex network of couplings.

This initial result shows that, on average, OSS developers actively use source folders as containers of similar-scoped elements, and prefer linking elements in the same folder rather than coupling different folders. However, this result should not be used to statically judge a software system; the Arla system is not inherently worse than the others analysed, but on average its source folders are more instable, as per the definition given. Based on that, it is likely that selecting reusable folders from this system will be more difficult.

3.2 Evaluation of Reuse and Architectural Properties – Gaim

The trend of Gaim's folders growth has a stabilisation phase only in the middle part of its lifecycle. It is possible to conclude that MPlayer has achieved a mature status similar to XMMS, while Gaim is still on a fully development stage. The key findings in the case of Gaim are as follows:

1. External libraries: this folder in the latest release of Gaim has an afferent coupling of 26 (out of 32 overall) folders, but no efferent coupling. This confirms that, from a coupling perspective, the "libc" folders is highly reusable, and its instability is minimum.
2. International folder: Gaim incorporates the "intl" folder, albeit not from the first release, and this folder behaves in a similar way to that observed below in the XMMS system. Again, this confirms it as a reusable asset, based on a coupling perspective.

3.3 Identifying Reusable Folders – Dynamic Analysis

In this section, the data gathered in the evolutionary exploration of the four case studies will be used to extract reusable folders. In particular, the coupling patterns of the "libc" and the "International" folders will be looked for in other folders. Low values in the instabilities will trigger the definition of reusability of the folder, and a preference will be given to folders with larger extensibility. Tentatively, two thresholds were set: the

joint combination of an instability lower than 0.2, and an extensibility larger than 10, highlight a folder as reusable.

In table 1, a list of reusable folders per project is given, based on the instability and extensibility factors, defined above. The following points are relevant to interpreting the columns of the table:

1. Each project has a set of rows, pointing at reusable folders (second column) found in that project. In each set, folders with a small instability (3rd column) and containing a larger amount of source files (i.e. higher extensibility, 4th column) are preferred as potential reusable folders.
2. Efferent coupling (5th column) has been evaluated via the product of the number of efferent folders and the number of total efferent calls. Afferent coupling (6th column) is given by a similar product, but involving afferent folders and calls. Intra-folder calls are summarised by the "Calls to self" column, while links to the "Libc" folder are shown in the "Calls to libc" column. In many cases, the amount of intra-folder calls are much more than the amount of efferent calls, which confirms the lowest instability of these folders.
3. A description of each folder (last column of table 1) has been determined either from the description files contained in the folder, or by browsing the documentation. This task is of key importance in order to describe a folder to potential reuses, and it has not been possible to automate this task.

Validation of the predictors – Instability and Extensibility As stated above, and considering the relatively few empirical studies focused on the reuse of OSS components, the practice of reuse of OSS components is not widespread, and it needs further investigation. The implemented algorithm selects folders which are being actively reused by these systems (the "rx" folder, third row, provided by IBM, reused in the Arla system; the folder "tremor" in the MPlayer system). In terms of validation of the proposed metrics as predictors of external reusability, the following lists the approach used:

– **Detecting reused folders**: the list of reusable folders, as listed in table 1, has been processed in a semi-automatical way, through various engines: the main SourceForge site [2], the Krugle code search engine [3], and the FLOSSmole repository [4] have been searched against each of these folders. The SourceForge site has been searched manually, browsing for the names of each folder, and analysing whether new projects exist as a spin-off from that folder, or if existing projects include the requested fodler; the Krugle engine has also been manually searched, and the existing OSS projects that include the requested fodler have been detected; the FLOSSMole repository has been automatically searched for matching names of new projects with the name of the requested folder.
– **Detection of actual reuse**: the folders found in any of the information sources have been detected as such. No further analysis has been performed to check whether

[2] http://sourceforge.net/
[3] http://www.krugle.com/
[4] http://ossmole.sourceforge.net/

their current coupling interaction, or their extensibility, has changed overtime as the original case studies.

Based on the approach above, it was found that some of the highlighted folders are currently distributed as *independent* OSS projects:
– the “liba52” folder of MPlayer (7th row of table 1, and
– the “libxmms” folder of XMMS (12th row of table 1).

In terms of “external” reusability [32], it was also found that some of the folders in the MPlayer project are reused in various OSS projects:
– the “liba52” folder is currently reused by the “gst-ffmpeg” project;
– the “libavcodec” folder is currently reused by several other OSS projects (“gst-ffmpeg”, “xmovie”, “quicktime4linux”, “mythtv” among others);
– the “libfaad2” folder is reused by the “audacious-plugins” project;
– the “libmpdemux” folder is reused by the “nmm” project.

False Positives The algorithm as illustrated above is subject to detect false positive, i.e. targetting folders as reusable but never reused. Based on the given thresholds (Instability <= 0.2; Extensibility >= 10), the latest analysed releases of the analysed projects presented the false positives listed in the last column of table 1. One could gather these false positives into two main categories, the first contains those folders which currently represent most of the functionalities of the system (*e.g.* the “root/src” folder in Gaim, and the “root/xmms” in the XMMS system). These cases are typically large-grained components, and in terms of reusability, they should be split into other components before being reusable. The second category contains those other folders which present a reusability potential, but currently are not reused. It is the opinion of the authors that these false positives represent missed reuse opportunities.

4 Related Work

This work is related to various research areas: reuse of components, empirical studies on software systems, graphic visualizations, software couplings, and software architectures. Since this work is in a larger research context, related to the study of the evolution of OSS systems, from which the case studies presented in this paper have been taken, empirical studies of OSS are also relevant to this research. In the following Section, an overview of the related works is presented, and consideration is given to determining how this work expands upon the related work.

The research with the closest scope to the present work is presented in [20], where a framework is proposed for the reuse of components in the OSS environment. It points out some key aspects to consider carefully, and which could impede its implementation, such as the license types, the ego-boosting problems or the programming languages; these social aspects were previously stressed also in [34]. The technical aspects of incorporating external code are also mentioned, but no in-depth analysis is provided. The

present work studies some of the technical details of selecting reusable folders, but the mentioned aspects are all key points which should be given consideration as well.

As mentioned above, many *reuse* research studies (and a set of specific conferences on the topic of "Software Reuse") have been devoted to developing techniques [22, 37] and frameworks for globally enhancing reuse [3], establishing state-of-the-art and critical aspects of reuse [25, 30]. This present work has been conceived as having the OSS development communities as its main recipients and beneficiaries in order that results and techniques of this academic research can be fed back to the OSS communities and advance the development of their systems.

This work is also related to the study of *software architectures*: previous works ([18, 19, 38]) have defined and used different views of architecture of a software system. For example, [19] refers to a "4+1" view model to describe a system involving logical, process, physical, development views, and user scenarios. This model defines different perspectives for different stakeholders; the present work uses the concepts of logical ("hierarchical") and process ("coupling") views to establish a comparison between them. Similarly, [18] defines four architectural views of software systems, which in turn focus on coarser degrees of granularity (conceptual, or the abstract design level; module, or the concrete design level; code, or components level; and execution level). As stated above, the present research focuses on the views which are closer to the work of software developers, as, for instance, the folder or the file level. In the selection of attributes, the limit is on those that it is possible to derive from projects found in existing OSS repositories with a reasonable effort. Hierarchical ("abstract design level") and coupling ("component level") views can both provide insight into how developers deal with macro and micro-components of software systems, respectively.

Recently, it has been realized that empirical data for OSS systems is more widely available than that for proprietary systems. A general distinction can be drawn among these studies. In part, research studies are based "on" OSS systems "for" advancing the Open Source Software Engineering body of knowledge; other studies access OSS projects for generating boundary crossing conclusions on software systems in general. Recent studies of the first kind include those examining single OSS projects [2, 15, 16] [14, 35], or those examining several OSS projects [6, 7, 28]. This work is intended as a means to directly inform OSS developers of the availability of existing potentially reusable folders upon which they can build new applications.

As previously reported, recent work [11, 23] has been focused on OO *package analysis*, in order to characterise the roles of specific folders. This work is greatly inspired by these research studies, and focuses the "source folder" as the fundamental unit in a network of couplings. The advances presented in this paper are based on considering interaction coupling within procedural languages as the most representative in an OSS context [7], on providing an evolutionary perspective of these interactions, and on focusing the analysis on the reusability of folders based on their couplings.

Recent work on *code couplings* in OSS has been reported in [1, 39], where the analysis used the definition of common coupling; two or more modules are commonly coupled when they share a reference to the same variable. Our approach is slightly different, since the source code (mostly C with some C++) is analysed by considering three different couplings (dependency, and include coupling, and calls among functions). We

consider their relevance from the point of view of two different visualizations, in order to define a relationship between code coupling and what we define as the folder structure of a software system at a given stage of its evolution. The definitions of coupling, as used throughout this paper are mirrored in those presented in [36]; specifically, the common coupling (in this paper calculated as file dependencies), and the control coupling (in this paper calculated as function calls).

5 Conclusions, Further Work and Threats to Validity

This paper has presented an approach to evaluate the source folders of a software system as potentially reusable and shareable fine-grained components. The current state of the art in terms of reusability are two-fold: the commercial internal reuse, which is typically not shared, and the COTS approach, which reuses "black-box" components.

This paper focused its reuse approach on smaller components, the folders (or directories) of a software system. Building on the vast amount of OSS knowledge and the OSS code base, specific source folders were observed as successfully reused across OSS systems. An analysis of the coupling (i.e., the interactions among various other folders) was carried out in order to characterise these specific folders based on patterns of interaction. The approach described above had two objectives. The first objective was to look for similar coupling patterns in other folders in order to identify potential candidates for reuse in other OSS projects. The second was to identify actions that developers should consider in order to improve the reusability of folders of their project for other OSS projects.

Regarding the first objective, the empirical results are based on literature definitions. It was found that successfully reused folders have a low instability index, i.e., they provide more services to other folders than they ask for from other folders. In a service-based terminology, these folders act mostly as servers for other folders. This coupling pattern was searched for in other source folders, and a list of folders with a similar behaviour was provided in table 1; these folders represent potentially reusable components. In terms of external reusability, the algorithm identified some source folders which are already being reused in the OSS community as side projects of existing OSS systems.

Various areas are being evaluated as further work: a key aspect of this research that should be enhanced is the extraction of information to characterise the potentially reusable source folders; this should be made automatic and non-invasive. Then, as exposed above, other types of coupling (dynamic and data couplings, inheritance etc.) have been identified in previous works, and should be considered to provide a more complete picture. Finally, it is planned to use a tooling technique to bind and/or resolve external dependencies: we wish to explore whether even modules with many dependencies could be highly reusable.

Threats to validity have been identified in the following aspects:

1. The usage of instability and extensibility alone could not be enough to categorize a source folder as reusable. Due to transitive dependencies, developing a new module using others, it will automatically becomes less reusable than the ones that were

reused (because Ce increases), unless it was manage to create many dependencies to the new module (such that Ca increase as well).
2. Only the dependency, inclusion and function calls couplings are studied. Other types such as data coupling [5], or dynamic coupling, [1]), are not considered. Further works will enhance our analysis to consider these types, and could bring more insights into these types of coupling.
3. Other characteristics determine whether a module should be reused in another system. Apart from those already cited by [20], there could be inherent reasons for not reusing a specific module, even if its instability is low at the coupling level. It could be that it is too small, or that it is very complex (in terms of cyclomatic complexity, for instance).

References

1. Arisholm, E., Briand L.C. and Foyen, A.: Dynamic Coupling Measurement for Object-Oriented Software. IEEE Transactions on Software Engineering, 30(8):491–506, 2004.
2. Aoki, A., Hayashi, K., Kishida, K., Nakakoji K., Nishinaka Y., Reeves B., Takashima A., and Yamamoto, Y.: A case study of the evolution of jun: an object-oriented open-source 3d multimedia library. In Proceedings of the 23rd International Conference on Software Engineering, pages 524–533, Toronto, Canada, 2001. ICSE.
3. Basili, V.R. and Rombach, H.D.: Support for Comprehensive Reuse. IEEE Software Engineering Journal, 6(5):303–316, 1991.
4. Beecher, K., Boldyreff, C., Capiluppi, A., and Rank, S.: Evolutionary Success of Open Source Software: an Investigation into Exogenous Drivers. Electronic Communications of the EASST: ERCIM Symposium on Software Evolution, 17(8), 2007.
5. Briand, L.C., Morasca, S. and Basili, V.R.: Property-based Software Engineering Measurement. IEEE Transactions on Software Engineering, 22(1):68–86, 1996.
6. Capiluppi, A: Models for the Evolution of OS Projects. In Proceedings of the International Conference on Software Maintenance, 65–74, Amsterdam, Netherlands, 2003.
7. Capiluppi, A., Lago, P. and Morisio, M.: Evidences in the Evolution of OS Projects Through Changelog Analyses. In Proceedings of the 3rd Workshop on Open Source Software Engineering, Portland, OR, USA, 2003. ICSE.
8. Capiluppi, A., Morisio, M., and Ramil, J.F.: Structural Analysis of Open Source Systems. In N. H. Madhavji, J. F. Ramil, and D. Perry, editors, Software Evolution and Feedback: Theory and Practice, pages 207–222. Wiley, 2006.
9. Capiluppi, A., Morisio, M. and Ramil, J.F.: The Evolution of Source Folder Structure in Actively Evolved Open Source Systems. In Proceedings of the 10th International Software Metrics Symposium, pages 2–13, 2004.
10. Carney, D.: Assembling Large Systems from COTS Components: Opportunities, Cautions, and Complexities. Technical report, SEI Monographs on the Use of Commercial Software in Government Systems, 1997.
11. Ducasse, S., Lanza, M. and Ponisio, L.: Butterflies: A visual Approach to Characterize Packages. In Proceedings of the 11th International Software Metrics Symposium, 2005.
12. Ellson, J., Gansner, E., Koutsofios, L., North, S.C. and Woodhull G.: Graphviz, Open Source Graph Drawing Tools, 2002.
13. Fenton N.E. and Pfleeger, S.L.: Software Metrics: a Practical and Rigorous Approach. Thomson, 1996.
14. Koch, S. and Schneider, G.: Effort, Cooperation and Coordination in an Open Source Software Project: GNOME. Information Systems Journal, 12(1):27–42, 2002.

15. German., D. M.: Using Software Trails to Reconstruct the Evolution of Software. Journal of Software Maintenance and Evolution: Research and Practice, 16(6):367–384, 2004.
16. Godfrey, M.W. and Tu, Q.: Evolution in Open Source Software: A Case Study. In Proceedings of the International Conference on Software Maintenance, pages 131–142, San Jose, CA, USA, 2000.
17. Gorton I., and Zhu, L.: Tool Support for Just-In-Time Architecture Reconstruction and Evaluation: an Experience Report. In Proceedings of the 27th international conference on Software engineering, pages 514–523, 2005.
18. Hofmeister, C., Nord, R. and Soni, D.: Applied Software Architecture. AddisonWesley, 2000.
19. Kruchten, P.: The 4+1 View Model of Architecture. IEEE Software, 12(5):88–93, 1995.
20. Lang, B., Abramatic, J.F., Gonzalez-Barahona, J.M., Gomez, P., Pedersen, M.K.: Free and Proprietary Software in COTS-Based Software. Lecture Notes in Computer Science, 34(12):2, 2005.
21. Li, W. and Henry, S.: Object-oriented Metrics that Predict Maintainability. Journal of Systems and Software, 23(2):111–122, 1993.
22. Llorens, J., Fuentes, J., and Astudillo, H.: Incremental Software Reuse. In Proceedings of the International Conference on Software Reuse, Torino, Italy, 2006. ICSR.
23. Lungu, M., Lanza, M. and Girba, T.: Package Patterns for Visual Architecture Recovery. In Proceedings of the Conference on Software Maintenance and Reengineering, 32–41, 2006.
24. Martin, R.C.: Agile Software Development, Principles, Patterns, and Practices. Prentice Hall, October 2002.
25. Matsumoto, Y.: Some Experience in Promoting Reusable Software Presentation in Higher Abstraction Levels. IEEE Transactions on Software Engineering, 12(1):43–60, 2004.
26. McClure, C.: Software Reuse Techniques. Prentice-Hall, 1997.
27. Medvidovic, N. and Jakobac, V.: Using Software Evolution to Focus Architectural Recovery. Automated Software Engineering, 13(2):225–256, 2006.
28. Mockus, A., Fielding, R. T. and Herbsleb, J. D.: Two Case Studies of Open Source Software Development: Apache and Mozilla. ACM Transactions on Software Engineering and Methodology, 11(3):309–346, 2002.
29. Mohagheghi, P. and Conradi, R.: Different Aspects of Product Family Adoption. In Proceedings of 5th International Workshop on Product Family Evolution, pages 429–434, 2003.
30. Morisio, M., Ezran, M. and Tully, C.: Success and Failure Factors in Software Reuse. IEEE Transactions on Software Engineering, 28(4):340–357, 2002.
31. Morisio, M., Seaman, C.B., Parra, A.T., Basili, V.R., Kraft, S.E. and Condon, S.E.: Investigating and Improving a COTS-based Software Development. In Proceedings of International Conference on Software Engineering, pages 32–41, 2000.
32. Poulin, J.S.: Measuring Software Reuse: Principles, Practices, and Economic Models. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
33. Pressman R.S.: Software Engineering: a Practitioner's Approach (2nd ed.). McGraw-Hill, Inc., New York, NY, USA, 1986.
34. Senyard, A. and Michlmayr, M.: How to Have a Successful Free Software Project. In Proceedings of the 11th Asia-Pacific Software Engineering Conference, pages 84–91, Busan, Korea, 2004. IEEE Computer Society.
35. Stamelos, I., Angelis, L., Oikonomou, A., and Bleris, G. L.: Code Quality Analysis in Open-Source Software Development. Information Systems Journal, 12(1):43–60, 2002.
36. Stevens, W.P., Myers, G.J. and Constantine, L.L.: Structured Design. IBM Systems Journal, 13:115–139, 1974.
37. Torchiano, M. and Morisio, M.: Overlooked Aspects of COTS-based Development. IEEE Software, 21(2):88–93, 2004.

38. Tu, Q. and Godfrey, W. M.: The Build-Time Software Architecture View. In Proceedings of 2001 International Conference on Software Maintenance, pages 65–74, Florence, Italy, 2001. IEEE.
39. Yu, L., Schach, S.R., Chen, K. and Offutt J.: Categorization of Common Coupling and Its Application to the Maintainability of the Linux Kernel. IEEE Transactions on Software Engineering, 30(10):43–60, 2004.

Folder	I	E	wCe	wCa	Calls to self	Calls to libc	Description	False positive
Arla project – reusable folders								
root/lib/roken	0.01	145	11*73	71*1326	395	367	Library handling missing or broken parts	yes
root/rx	0.03	33	13*111	53*1042	495	141	Library implementing the rx protocol	no
Gaim project – reusable folders								
root/src	0.030	139	12*1117	30*14462	9437	868	Common source files of the Gaim system	yes
MPlayer project – reusable folders								
root/liba52	0.02	23	2*21	9*230	196	15	ATSC A/52 stream decoder	no
root/libavcodec	0.03	154	22*155	27*4493	3780	168	Library for coding and decoding video and audio streams	no
root/libaf	0.03	38	7*52	25*464	294	112	Audio filter layer library	no
root/libfaad2	0.06	86	6*23	2*1063	1096	37	Decoding library for AAC formats	no
root/libmpdemux	0.06	136	21*257	28*2852	2740	353	Demultiplexer Library for MPEG, ASF, AVI formats	no
root/tremor	0.045	29	4*17	4*363	372	34	Tremor integer-only Ogg Vorbis audio codec	no
root/loader	0.054	27	8*203	24*1192	688	75	N/A	yes
root/osdep	0.066	25	6*47	23*174	17	54	N/A	yes
root/loader/wine	0.138	27	5*142	14*318	83	11	Header files for the Microsoft Windows compatibility	yes
XMMS project – reusable folders								
root/libxmms	0.003	19	4*20	25*1081	416	86	Generic library for the XMMS project	no
root/xmms	0.091	87	20*277	26*2121	2157	121	Common source files of the XMMS system	yes

Table 1. Reusable folders detected via the coupling analysis: wCe refers to the product “Efferent folders * Efferent calls”, while wCa refers to a similar product of afferent folders and calls