Software Engineering in Practice: Design and Architectures of FLOSS Systems

Andrea Capiluppi and Thomas Knowles

Centre of Research on Open Source Software,
University of Lincoln,
Brayford Campus, Lincoln,
LN5 7TS, United Kingdom
{acapiluppi,tknowles}@hemswell.lincoln.ac.uk

Abstract. Free/Libre/Open Source Software (FLOSS) practitioners and developers are typically also users of their own systems: as a result, traditional software engineering (SE) processes (*e.g.*, the requirements and design phases), take less time to articulate and negotiate among FLOSS developers. Design and requirements are kept more as informal knowledge, rather than formally described and assessed. This paper attempts to recover the SE concepts of software design and architectures from three FLOSS case studies, sharing the same application domain (i.e., Instant Messaging). Its first objective is to determine whether a common architecture emerges from the three systems, which can be used as shared knowledge for future applications. The second objective is to determine whether these architectures evolve or decay during the evolution of these systems. The results of this study are encouraging: albeit no explicit effort was done by FLOSS developers to define a high-level view of the architecture, a common shared architecture could be distilled for the Instant Messaging application domain. It was also found that, for two of the three systems, the architecture becomes better organised, and the components better specified, as long as the system evolves in time.

1 Introduction and Related Work

During the last years Free/Libre/Open Source Software (FLOSS) has gained much attention in the SE research community. This is due to various reasons, ranging from the availability of the software products, to the archival of past software and non-software artifacts in versioning repositories (bug tracking systems and mailing lists, among others). Two main types of FLOSS literature have been observed since, one termed *external* and the other *internal* to the FLOSS phenomenon [6]. Based on the availability of FLOSS data, the former has traditionally used FLOSS artefacts in order to propose models [17], test existing or new frameworks [9,20], or build theories [3] to provide advances in the more general SE field. The latter instead includes several other studies that have analyzed the FLOSS phenomenon *per se* [10,14,16,26], with their results aiming at both building a theory of FLOSS, and characterizing the results and their validity specifically as inherent to this type of software and style of development.

While much empirical research has used FLOSS as case studies where data and experience can be easily obtained, FLOSS-specific issues have also been identified.

C. Boldyreff et al. (Eds.): OSS 2009, IFIP AICT 299, pp. 34–46, 2009.

Generically, it should be established whether well-established approaches, techniques, frameworks and tools from the traditional SE knowledge apply to FLOSS practitioners and developers [2]. Specifically, FLOSS systems and their architectures have attracted significant attention among researchers due to the distributed constraints, modularity and the issue of collaboration. Developers sporadically joining FLOSS projects do not always have a clear understanding of the underlying architecture, and may break the overall conceptual structure by several small changes to the code base [27].

Also, it has been suggested that core FLOSS developers might hinder improvements to the software architecture to protect their privileged positions, thus precluding the system's future development [7]. In other words, the system's architecture could either be lost or *decay* over time.

Past SE literature has firmly established that software architectures and the associated code decay over time [13], and that the pressure on software systems to evolve in order not to become obsolete plays a major role [19]. As a result, software systems have the progressive tendency to loose their original structure, which makes it difficult to understand and further maintain them [24]. *Architectural recovery* is one of the recognized counter-measures to this decay [12]. Several earlier works have been focused on the architectural recovery of proprietary [12], closed academic [1], COTS [5] and FLOSS [8,15,27] systems; in all of these studies, systems were selected in a specific state of evolution, and their internal structures analyzed for discrepancies between the *conceptual* and *concrete* architectures [27]. Repair actions have been formulated as frameworks [23], methodologies [18] or guidelines and concrete advice to developers [27].

From these previous empirical reports and from specific calls for research on FLOSS architectures and their decay [4], the following open research questions have been formulated for the research reported in this paper:

1. Even if not specifically imposed by FLOSS developers, does any common system architecture emerge from FLOSS projects sharing the same application domain?
2. Is it possible to assess whether this architecture decays or improves during the evolution of a FLOSS project?

In order to tackle these questions, this research evaluates three FLOSS systems (Ayttm[1], Miranda[2] and Pidgin[3]), sharing the same application domain (Instant Messaging – IM), and compares the evolution of their conceptual, hierarchical views [25] with the architectural view based on the common coupling among components. As stated above, the objective is two-fold; at **first**, a common architecture will be sought, encompassing the three case studies. This could help other FLOSS developers within the same application domain to comply with the design notion and the shared (tacit) knowledge of the domain of other FLOSS developers. The **second** objective will study whether the architecture of these systems decays or not in their evolution and maintenance.

This paper is articulated as follows: Section 2 introduces the case studies, the two visualisations used throughout the paper, and how they were extracted, and presents the

[1] `http://ayttm.sourceforge.net/`
[2] `http://www.miranda-im.org/`
[3] `http://www.pidgin.im/`

case for the architectural decays of FLOSS systems. Section 3 summarises the main findings on the empirical analysis of the three projects, and determine a threshold for architectural decay, visualising the point where each system broke it, while Section 4 concludes and pinpoints future works.

2 Empirical Approach

This section details the empirical approach used throughout this research. At first, the rationale for the selection of the three systems will be given, next an overview of the extraction of the raw data into results will be outlined. In particular, attention will be given to the hierarchical [25] and concrete [27] architectures.

The selected FLOSS projects were chosen for the following reasons:

- There are similarities in their basic functions, since they all are "multi protocol instant messaging clients": no previous empirical study has tried before to detect the shared architecture of similar-scoped FLOSS systems;
- There are similarities in the underlying programming languages (Ayttm and Pidgin are implemented in C, and Miranda in C++);
- They are long-lived, established FLOSS projects (hence all their software is available), with an established community of developers and users: as visible in Table 1, Ayttm is an ongoing project since 2002, Miranda since 2000, and Pidgin since 1999.

2.1 Hierarchical and Common Coupling

A posssible method of examing the architecture of the software is through the nesting of folders, and their relations of "contained-in" results in the so-called *treestructure* [25]. Observing the disposition of source code within folders (i.e., *source folders*) can give potential developers the initial insight of how code is organised within folders [22]. The tree structures have been extracted for the three temporal points of each system (first available, latest available and middle releases), and the number of "nodes" (i.e., the source folders) of the corresponding tree has been recorded in Table 1. One such example, for the Pidgin system, is given in Figure 1: the dashed lines represent the hierarchical structure of folders contained in higher level folders.

On the other hand, in order to produce an accurate description of the concrete architecture suggested by [27], each project has been parsed using Doxygen[4]. The following notation was used:

- **Coupling:** This is the union of all the includes, dependencies and functions calls (i.e., the common coupling) of all source files as extracted through the Doxygen source code documentation generator tool. The *file-to-file* couplings were converted into *folder-to-folder* couplings, considering the folder that each of the above files belongs to; coupling among functions and/or methods was also converted into folder-to-folder coupling. A stronger coupling link between folder A and B would be found when many items within A call items of folder B (or viceversa). Couplings are depicted in Figure 1 as directed arrows, the amount of actual calls being summarised by the number on the arrow.

[4] `http://www.doxygen.org/`

- **Connection:** Distilling the couplings as defined above, one could say, in a boolean manner, whether two folders are linked by a *connection* or not, despite the strength of the link itself. The overall number of these connections is recorded in Table 1: the connections of a folder to itself are not counted (for the encapsulation principle), while the two-way connection $A \rightarrow B$ and $B \rightarrow A$ is counted just once (since we are only interested in which folders are involved in a connection). In Figure 1 it is possible to count 5 connections ($src \rightarrow irc$, $src \rightarrow msn$, $src \rightarrow oscar$, $src \rightarrow napster$ and $msn \rightarrow oscar$). Connections to the external libraries ("EXT_LIB") are not counted, since only the internal architectural properties of these systems are studied.

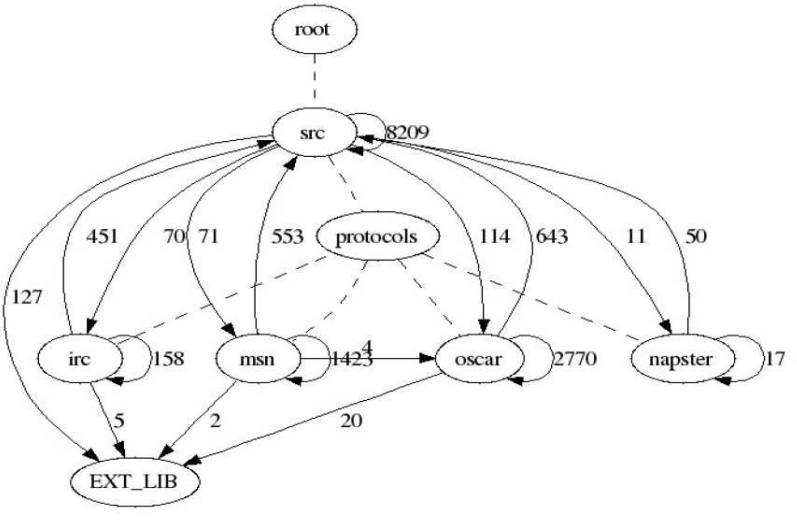

Fig. 1. Pidgin – tree structure, couplings and connections

Table 1. Summary of metrics

	Ayttm	**Miranda**	**Pidgin**
Active since	2002	2000	1999
Releases studied	21	35	114
Nodes (initial)	21	4	1
Nodes (middle)	32	44	33
Nodes (final)	32	72	50
SLOCs (initial)	70,493	3,692	5,679
SLOCs (middle)	103,044	118,641	152,370
SLOCs (final)	99,572	383,146	297,471
Connections (initial)	47	3	1
Connections (middle)	125	168	90
Connections (final)	122	390	223

2.2 Methods of Metrics Extraction

Source code for each case study was extracted and Doxgen was used on each version of the software. DOT [5] files (mapping the connections between files) are generated by Doxygen and several PERL scripts were applied to DOT files to extract the couplings and connections of each file/module. Using the data regarding the connections between files a hierarchical map of the software was possible as seen in figures 1 - 5.

3 Results

3.1 Common Instant Messaging (IM) Architecture

During the analysis of the evolution of the studied systems, recurring patterns have been observed in the naming of folders containing source code, and the directions and frequency of connections among them.

1. **Core.** The first set of folders comprises the core function of an IM client. Connection to IM networks, handling of IM contacts, drawing of the underlying GUI are all examples of the "core" functions of an IM system. Most of its connections are handled from and to elements (files, functions) contained within it, while links to other components see this component acting as a"server" of functions, rather than a receiver [11].
2. **Protocols.** The second observed cluster of folders, attracting a considerable amount of couplings, deals with the supported IM protocols (e.g., Yahoo, Jabber, etc.). Since the very early releases of each system, several calls are directed from and to folders named after each protocol. A container "protocol" folder, keeping source code shared by several protocols, is also commonly found, while some protocols can be further expanded in other exportable folders (e.g., libyahoo, libjabber, etc.).
3. **Plugins.** The third component comprises the plugins handled by the IM client, ranging from GUI skins managers to connectors to email clients. A "plugin" umbrella folder is also used for the same purposes as the "protocol" above. The main purpose of this component is to hold those functionalities which are not considered as fundamental for an IM client. In terms of connectors, each plugin can be considered as a stand-alone system, typically linked with few couplings to other plugins and other parts of the IM system.

In summary, a common architecture of IM systems has emerged, as visible in Figure 2. The three basic components (core, plugins and protocols) appear as primitives in the early releases of the observed systems, and get better refined and modularized during the system's lifetime, meaning that the refined components send most of their calls to elements within themselves. Each component exists as a cluster of several source folders; the hierarchical structure [25] comprises umbrella folders ("core", "plugins", "protocols") containing other folders, typically at the same level of nesting, acting as placeholder for source files of specific modules ("aycryption", "ticker", "yahoo", can be seen as an example module contained in each component).

[5] `http://www.graphviz.org/`

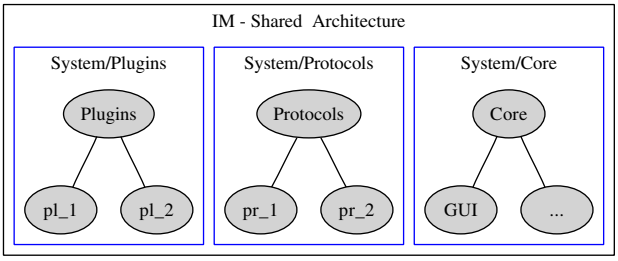

Fig. 2. Shared IM architecture

3.2 Evolution and Architectural Decay

The very early releases of the studied systems show on average few folders (see Table 1) with the exception of Ayttm, which appears well modularized already in the first available release (21 folders). On the other hand, the latest studied releases show projects which have been extensively decomposed into folders and subfolders. Throughout their evolution, as new folders get connected into the system, new couplings are introduced, in part within the components of the three-tier architecture described above, and in part connecting the components among each other. This second type of couplings taints the desirable independence of an architectural component, making it a requester or a server of services from other components.

In this Section, the architecture of the three systems, as proposed above, is observed in three temporal points, namely the earliest and the latest available releases, and the release in between the first two. The objective of this analysis is to observe how the systems have evolved with respect to their architectural components and connectors, and to draw instructive conclusions and guidelines for FLOSS developers. The overall evolution in number of source folders has also been studied. The number of nodes containing source code have been compared with the number of connections among them to give an insight into on how FLOSS developers counteract the increasing decay of their systems.

Ayttm. The top left part of Figure 3 shows the evolution in the number of source folders for the Ayttm project. Its initial state was already quite formed and structured, then a steep growth brought it to its current size and structure. These lasted up to the recent, scattered releases, which did not change significantly the internals of the project. In terms of the connections among different source folders, a new source folder added to the system brought 7 new connections with existing folders. This is evaluated excluding the connections tying elements within the same folder, and counting a single connection even when two folders are linked forward and backward (i.e. both act as server and requester of services to each other).

In Figure 3 the remaining graphs (in order, top right, bottom left and bottom right) show the evolution of the basic architecture (the components, and their connectors) in the three selected releases: only two components are detectable (“Core” and “Protocols”) in this system, while the hierarchical structure does not suggest the presence of

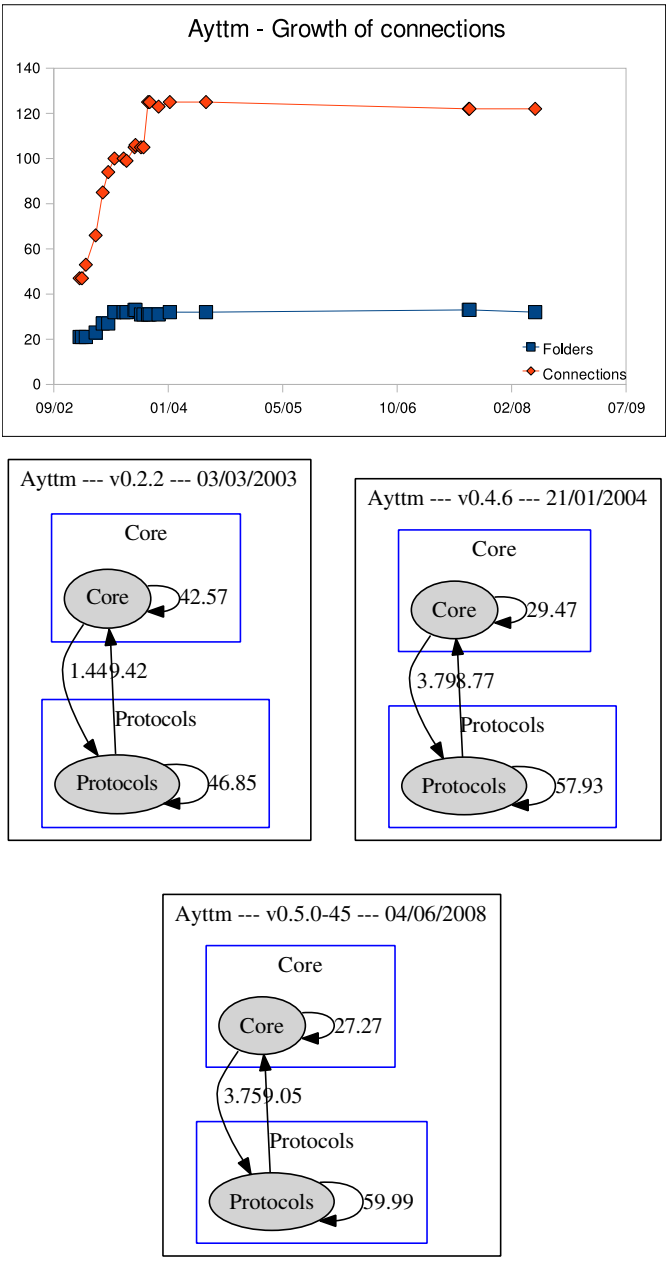

Fig. 3. Results (Ayttm) – Evolution of nodes, connections and architecture

the third component. Even so, the overall structure, and the balance of the couplings remains quasi-constant in the three points: there is still a severe dependency between the two components, each acting both as requester and server of services from the other.

Miranda. Similarly to what has been seen for the Ayttm system above, the top left part of Figure 4 shows how the number of source folders grew in Miranda: very few folders formed the initial releases of the system, which also counted on very few couplings among them. The 5th available release had a steep increase of functionalities, adding 26 new source folders, 111 new connections, but a decrease of 2,000 SLOCs, qualifying as a restructure of the system. As also visible in the plot, there are several releases which act as refinement of the current architecture, where decreases in the number of source folders are often paired to decrease in the couplings between folders. Every new source folder added to the system on average 6 new connections.

The other graphs of Figure 4 show the percentage of couplings among the components of the Miranda system in the three selected evolutionary points. As visible on the top right part, the Miranda system at its inception was just encapsulating the basic functionalities ("Core") and handling a selected number of IM protocols. Overall, the principle of independence permitted to have around 90% of the couplings within the same component, but the "Core" acted as a requester of services in 10% of the couplings (35 out of 345). In the bottom left and right graphs, the Miranda system developed a "Plugin" component, and the "Core" component reduced its requests to just 3% of the overall amount of couplings (358 calls), while the latest available release is even more modularized, having just 1,12% of the couplings being outward.

Pidgin. Finally, the top left part of Figure 5 shows the growth of source folders in Pidgin. Initially just one folder contained the source code of the first release. Unlike Ayttm, this system had a steady pace in its public releases, apart from a long hiatus during 2005 and 2006, when the original name of the project (Gaim) was changed due to copyright issues. Unlike Miranda, no major restructuring has taken places yet, but several small and medium decreases in both the number of folders and couplings between them have been observed. This continual maintenance effort has an effect also on the new folders being added; for each new folder, around 4 new connections have to be made on average with existing folders.

As done above, the three remaining graphs of Figure 5 show the architecture of this system and its components and connectors. Since its inception this system (although with just one folder containing the source code) already provided the three basic components which appear in each subsequent release. These primitives still have serious dependencies with each other which are later corrected and ultimately resolved in the latest available release (bottom right of Figure 5). The same component appears further decomposed into three subsystems, namely the "pidgin" graphical environment, the "libpurple" set of core functions, and the "finch" text-based version of the IM client.

From the reuse perspective, the approach described above is particularly relevant and conclusive for this system: the reuse of its components (or modules) within other systems is further simplified by how the connections have been designed and simplified throughout the lifecycle of this application. As reported above, the "core" of this system, in its latest release, shows three modules ("Finch", "Libpurple" and "Pidgin") which are independent from both the protocols and the plugins components: when in need of recreating a new IM client, developers could safely extract the "libpurple" module (responsible for the vast majority of the basic functionalities of an IM system) and reuse it as the basis of a new IM system. In fact, this module acts as a pure server, and does not rely on any other components or modules of the system in which it belongs.

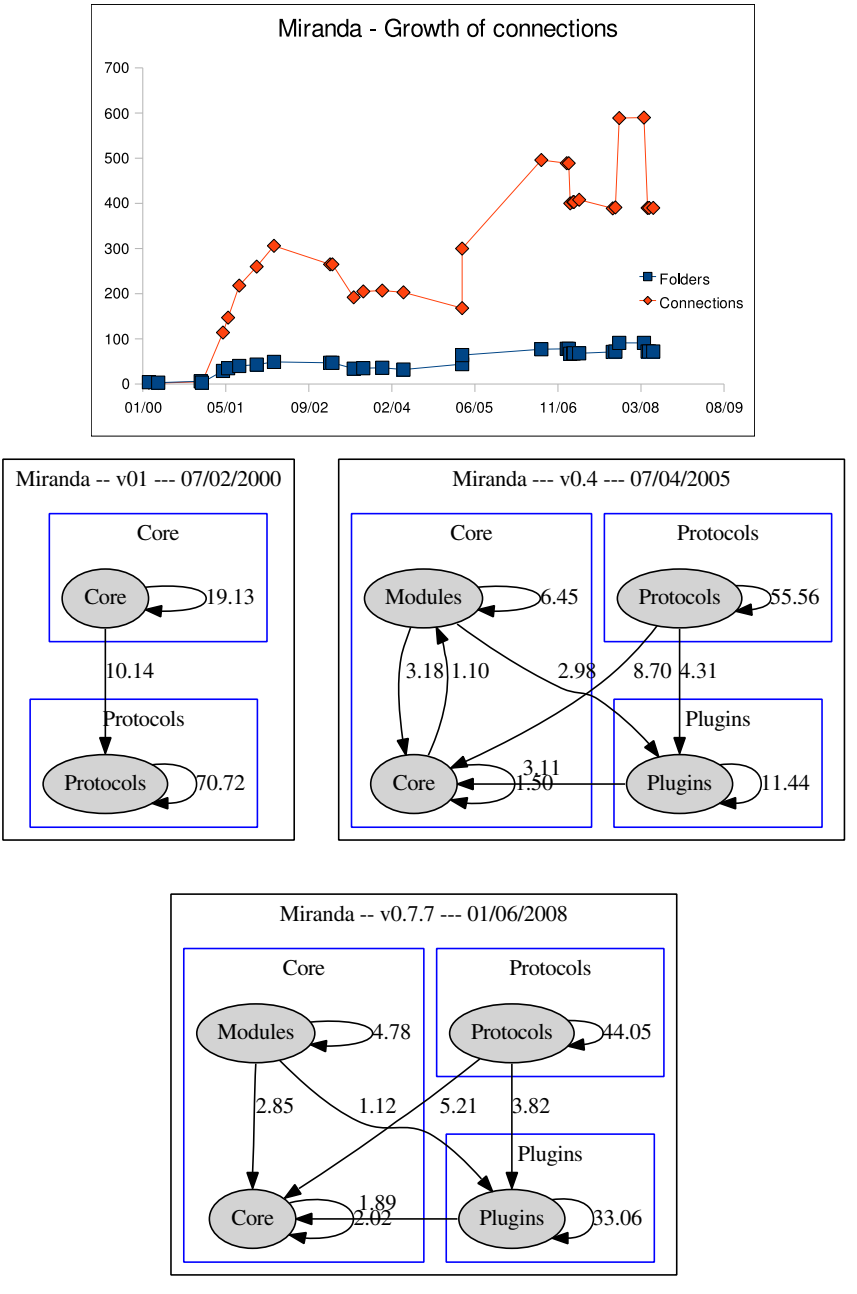

Fig. 4. Results (Miranda) – Evolution of nodes, connections and architecture

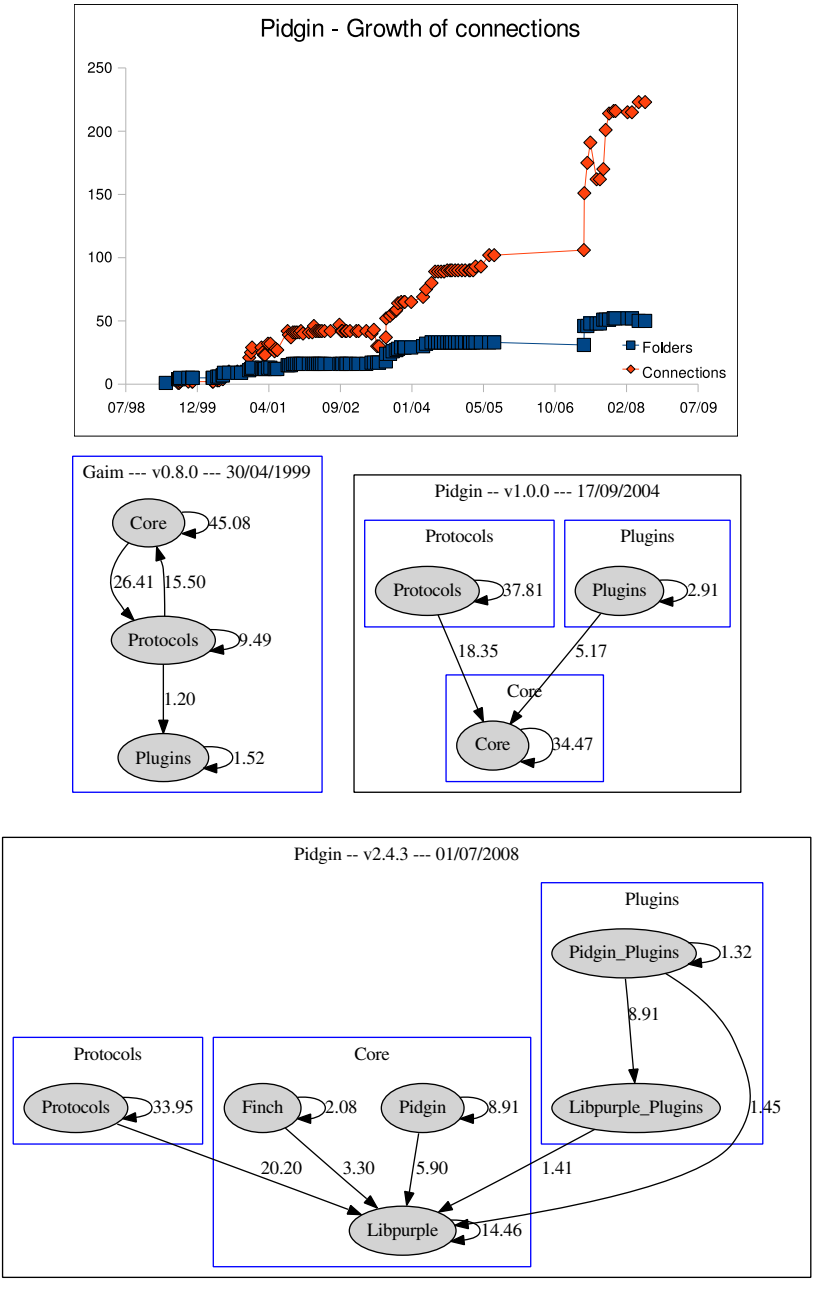

Fig. 5. Results (Pidgin) – Evolution of nodes, connections and architecture

4 Conclusions

This paper has attempted to make use of established Software Engineering concepts (software design and architectures) within the FLOSS development paradigm. It has been reported in the literature that established processes as the software requirements or the designs are written down or formalised by FLOSS developers, but instead those are tacitly understood by both developers and users.

Software architectures of FLOSS projects represent an active and open research field, since the distributed style of development of the FLOSS approach makes it easier for occasional developers to break the underlying assumptions of components and connectors, and their desirable independence. Past literature has already studied the architectures of FLOSS systems, but no longitudinal study has been performed yet, nor has the common architecture of several FLOSS projects, sharing the same application domain, been extracted.

This paper has presented the empirical observation of three FLOSS systems implementing the same functionalities and domain, with the aim of both extracting their tacitly agreed architectures, if any, and checking whether a common architecture could be formalised for the domain "Instant Messaging". A longitudinal approach was used to show the evolution of nodes and connectors in the three systems, and recurring components were sought.

It was found that a common architecture is currently at the base of the three studied systems, based on three components: a "core" component encompassing the basic functionalities of an IM client (managing network connections, handling contacts, drawing of the graphical interface, etc); a "protocol" component, managing the diverse protocols an IM client can support (AIM, Jabber, Yahoo, etc.); and finally a "plugin" component, whose objective is to enhance the basic functions with widgets not considered as core, similar to what is found in the "plugin" package of other systems (e.g., Sun's Eclipse).

It was also shown that these systems decay in terms of the underlying structure, since it was found that the addition of new folders and functionalities typically adds more than one connections to existing folders, making the overall understanding of the components more complicated. In two of three systems, though, these decay did not affect the overall architecture: in Pidgin and Miranda the three main components appear better and better defined, with a decreasing trend of connections outside the boundaries of the components themselves.

5 Threats to Validity

The use of the 2000 Holt paper for extracting the architectures of the software systems using the coupling method may be seen as out-dated in comparison to other newer techniques such as the Focus Method[21]. Additionally using one method to extract all of the architectures may also limit the variation of results.

Due to the domain of the software (IM), particular functionality (protocols) of the software are developed and structured in an exogenous manner in comparison to other modules. Protocols such as MSN, AOL or Yahoo must function and interact in a standardised way, which affect how other modules of the software are designed and developed. Due to this common and standardised functionality shared between all case studies, each

of the systems may inherit architectures in regards to protocols used. The source code for each case study was extracted and Doxgen were used on each version of the software. Using the DOT files (mapping the connections between files) generated by Doxygen several PERL scripts were applied to DOT files to gather the statistics of the interactions of each of the files. Using the data regarding the connections between files a hierarchical mapping of the software is possible as seen in figures one through five.

6 Future Work

The future works we are planning are essentially two-fold: on one hand, it should be studied how developers contribute to the decay of the architectures, by enabling connections among elements which belong to separate components. On the other hand, other application domains should be studied: the analysis of several multi-media systems, for instance, should reveal internal and reused libraries (codec, demux, conversion of formats) which form clear-cut components; an analysis of several web-server should highlight the presence of modular components (e.g., "logging", "access control" and others). We are planning to study how diverse systems in both achieved the same functionalities, and how these are mirrored in the architecture. Also, these studies (and all the ones making use of public data) should be made available to the developers of the studied systems, to try and bridge the gap between traditional Software Engineering and Open Source Software Engineering.

References

1. Abi-Antoun, M., Aldrich, J., Coelho, W.: A case study in re-engineering to enforce architectural control flow and data sharing. Journal of Systems and Software 80(2), 240–264 (2007)
2. Adams, P., Capiluppi, A.: Bridging the gap between agile and free software approaches: The impact of sprinting. International Journal of Open Source Software and Process 2009
3. Antoniol, G., Casazza, G., Penta, M.D., Merlo, E.: Modeling clones evolution through time series. In: Proc. IEEE Intl. Conf. on Software Maintenance 2001 (ICSM 2001), Fiorence, Italy, pp. 273–280 (2001)
4. Arief, B., Gacek, C., Lawrie, T.: Software architectures and open source software – Where can research leverage the most? In: Proceedings of Making Sense of the Bazaar: 1st Workshop on Open Source Software Engineering, Toronto, Canada (2001)
5. Avgeriou, P., Guelfi, N.: Resolving architectural mismatches of cots through architectural reconciliation. In: Franch, X., Port, D. (eds.) ICCBSS 2005. LNCS, vol. 3412, pp. 248–257. Springer, Heidelberg (2005)
6. Beecher, K., Boldyreff, C., Capiluppi, A., Rank, S.: Evolutionary success of open source software: An investigation into exogenous drivers. Electronic Communications of the EASST 8 (2008)
7. Bezroukov, N.: A second look at the cathedral and the bazaar. First Monday 4(12) (1999), `http://www.firstmonday.org/issues/issue4_12/bezroukov/index.html`
8. Bowman, I.T., Holt, R.C., Brewster, N.V.: Linux as a case study: its extracted software architecture. In: ICSE 1999: Proceedings of the 21st International conference on Software Engineering, pp. 555–563. IEEE Computer Society Press, Los Alamitos (1999)

9. Canfora, G., Cerulo, L., Penta, M.D.: Identifying changed source code lines from version repositories. Mining Software Repositories, 14 (2007)
10. Capiluppi, A.: Models for the evolution of OS projects. In: Proceedings of ICSM 2003, pp. 65–74. IEEE, Amsterdam (2003)
11. Ducasse, S., Lanza, M., Ponisio, L.: Butterflies: A visual approach to characterize packages. In: Metrics 2005: Proceedings 11th International Software Metrics Symposium (2005)
12. Dueñas, J.C., de Oliveira, W.L., de la Puente, J.A.: Architecture recovery for software evolution. In: CSMR 1998 – Proceedings of the 2nd Euromicro Conference On Software Maintenance And Reengineering, pp. 113–120 (1998)
13. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A.: Does code decay? assessing the evidence from change management data. IEEE Transactions on Software Engineering 27, 1–12 (2001)
14. 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)
15. Godfrey, M., Eric, H.: Secrets from the monster: Extracting mozilla's software architecture. In: CoSET 2000: Proceedings of the 2nd Symposium on Constructing Software Engineering Tools (2000)
16. Herraiz, I., González-Barahona, J.M., Robles, G.: Determinism and evolution. In: Hassan, A.E., Lanza, M., Godfrey, M.W. (eds.) Mining Software Repositories, pp. 1–10. ACM, New York (2008)
17. Hindle, A., German, D.M.: Scql: a formal model and a query language for source control repositories. SIGSOFT Softw. Eng. Notes 30(4), 1–5 (2005)
18. Krikhaar, R., Postma, A., Sellink, A., Stroucken, M., Verhoef, C.: A two-phase process for software architecture improvement. In: ICSM 1999: Proceedings of the IEEE International Conference on Software Maintenance, p. 371. IEEE Computer Society, Washington (1999)
19. Lehman, M.M.: Programs, cities, students, limits to growth? Programming Methodology, 42–62 (1978) (inaugural lecture)
20. Livieri, S., Higo, Y., Matushita, M., Inoue, K.: Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In: ICSE 2007: Proceedings of the 29th International Conference on Software Engineering, pp. 106–115. IEEE Computer Society, Washington (2007)
21. Medvidovic, N., Jakobac, V.: Using software evolution to focus architectural recovery. Automated Software Engineering, 13 (2006)
22. Murphy, G.C., Notkin, D., Sullivan, K.: Software reflexion models: bridging the gap between source and high-level models. In: SIGSOFT 1995: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering, pp. 18–28. ACM, New York (1995)
23. Sartipi, K., Kontogiannis, K., Mavaddat, F.: A pattern matching framework for software architecture recovery and restructuring. In: IWPC 2000: 8th International Workshop on Program Comprehension, pp. 37–47 (2000)
24. Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., Yan, H.: Discovering architectures from running systems. IEEE Transactions on Software Engineering 32(7), 454–466 (2006)
25. Spinellis, D.: Code Reading: The Open Source Perspective. Addison-Wesley Professional, Reading (2003)
26. Stamelos, I., Angelis, L., Oikonomou, A., Bleris, G.L.: Code quality analysis in open-source software development. Information Systems Journal 12(1), 43–60 (2002)
27. Tran, J.B., Godfrey, M.W., Lee, E.H.S., Holt, R.C.: Architectural repair of open source software. In: IWPC 2000: Proceedings of the 8th International Workshop on Program Comprehension, pp. 48–59. IEEE Computer Society, Washington (2000)