A Core-Periphery-Legality Architectural Style for Open Source System Development

Alexander Lokhman, Tommi Mikkonen, Imed Hammouda
Department of Software Systems
Tampere University of Technology
P.O. Box 553, FI-33101 Tampere, Finland
{firstname.lastname}@tut.fi

Rick Kazman, Hong-Mei Chen
ITM Department
University of Hawaii
Honolulu, HI, USA
{kazman,hmchen}@hawaii.edu

Abstract—Despite the rapid rise of open source systems (OSS), it is unclear what architectural patterns enable the success of these systems. There is little guidance on architectural patterns for OSS development. Consequently, the creators of OSS projects have often come to their own solutions intuitively, via trial-and-error and design iterations. To fill this void, we study the architectural challenges encountered by OSS projects and identify architectural patterns that address these challenges. We propose an integrated architectural style that addresses two challenges for OSS success: 1) what architectural principles and patterns enable the active participation of a large, distributed community, and 2) how to manage OSS licensing issues that prohibit open source (and proprietary) software to mix freely. It is critical to address these issues together as architectural decisions impact on licensing issues, which have an impact on the long-term survival of the system. We contribute to OSS development by identifying, integrating and advancing existing work on OSS patterns and offer an integrated architectural style that addresses these challenges simultaneously thus guiding future OSS development. We validate our approach via empirical studies of successful OSS systems that have evolved to use this style, and report on a case study using the proposed architectural style for a new OSS project.

Keywords—open source systems, software architecture, architectural patterns, software licensing, legality pattern, architectural style, core-periphery, Metropolis model

I. INTRODUCTION

Organizations commonly create complex systems by combining contributions from Open source systems (OSS) communities as well as proprietary software, creating new OSS and new commercial systems [1], [2]. However, the development of OSS poses a puzzle for software engineering researchers: Why are some OSS tremendously successful – such as Linux, Apache httpd, Mozilla, Eclipse – while most languish in obscurity? We define “success” as the ability to grow – in terms of lines of code, features, and size of the community of developers – seemingly without bounds [3]. This study aims at identifying “architectural design decisions” for successful OSS development. We do so for two reasons: First, since architecture is the foundation of any complex software system, its design is critical to any successful system development [4]. The architecture defines the characteristics of the system at an abstract level, and supports, to varying degrees, the system’s quality attributes, such as availability, scalability, modifiability, or performance [5]. Second, there are many advantages of using proven design solutions [6]: they save time because the designer does not have to reinvent the wheel, they provide a proven foundation upon which to build and innovate, they have known quality attributes properties.

There are well known architecture methods, patterns, and tactics for traditional systems but few have been described specifically for OSS, even though the architectures of many OSS have been described [7]. As each OSS architecture has been described in isolation there is little guidance for how these architectures may be reused in other contexts. In addition, OSS has distinctive characteristics; these are peer-produced, crowdsourced systems where centralized planning and control of architecture is difficult. Moreover, OSS encounter unique software licensing challenges that are qualitatively different from those of proprietary software systems, but are critical to OSS success and survival. Specifically, license restrictions often limit OSS open source (and proprietary) software from integrating. These problems motivate our research.

We will describe the challenges encountered by OSS projects and how these challenges can be seen as architectural requirements for developing successful OSS. We identify existing architectural patterns for OSS based on these requirements and propose an integrated architectural style for OSS. In contrast to the concept of architectural pattern, an architectural style is a named collection of architectural design decisions that are applicable in a development context, constrain architectural design decisions within that context, and promote beneficial qualities [4]. Thus an architectural style is closer to the concept of a system of patterns [8]. Our proposed architectural style follows this definition by addressing two challenges for OSS success simultaneously: 1) what architectural development principles and patterns enable the active participation of a large, distributed community for OSS, and 2) how to manage complicated OSS licensing issues that prohibit different code bases (OSS and proprietary) from mixing freely.

Recent studies have investigated these issues, but in isolation. For instance [9] articulated many of the salient characteristics of crowdsourced systems, including OSS,
II. ARCHITECTURAL AND LEGAL REQUIREMENTS FOR OPEN SOURCE SYSTEMS

OSS is vast and varied—any software can be turned into open source by releasing it under an open source license, as defined by the OSI1. However, OSS share many characteristics [9]: 1) open teams, 2) mashability (or interoperability) 3) conflicting, unknowable requirements, 4) sufficient correctness, 5) unstable resources, 6) emergent behaviors, 7) focus on operations, and 8) continuous evolution.

OSS projects frequently reuse as much software as possible. Successful OSS must: 1) encourage and support parallel activities and innovativeness of a large group of independent developers; 2) provide a basic functionality upon which developers can build; 3) provide the fundamental mechanisms for achieving the important quality attributes of the system; and 4) provide the means for combining components, so that projects can build upon other projects.

To achieve the goals of peer-produced, crowdsourced OSS development, two critical challenges must be addressed simultaneously, creating unique architectural requirements: 1) support distributed development: The architecture must promote “anarchic collaboration through extensions while ... preserving centralized control over the interfaces” [14]; 2) manage numerous licensing/legal issues associated with OSS development that are irrelevant when developing proprietary software.

1) Widely distributed development: The architecture must provide the maximum opportunity for developers to make individual, loosely-coordinated contributions without adversely affecting each other. It must facilitate building a sustaining core of committers, surrounded by a larger group of contributors [15]. In such systems the social structure of developers is typically reflected in the architecture of the system, where a highly coupled core of components developed by the core committers is surrounded by a larger periphery of components (and developers) that exhibit low coupling [16]–[18]. This architectural structure must provide an environment for new forms of development, collaboration, and co-creation as described in the “Metropolis Model” [9].

2) Legal issues for software reuse: Open source licenses are special copyright licenses that make source code available under terms that allow for modification and redistribution without a payment to the original author. Numerous licenses exist, and since different licensing terms are in no way coordinated, they frequently conflict in their terms and conditions. Even the most common licenses are often incompatible with each other due to subtle differences, and software released under them cannot be combined arbitrarily. As an example, a software component under the terms of GPL v2 cannot be directly linked with another under the terms of the Apache v2 license. The reason is that GPL

1http://www.opensource.org/licenses/
software cannot be mixed with software that is licensed under the terms of a license that imposes stronger or additional terms, in this case the Apache license. The license conflict between two open source components may be asymmetric. For instance, Apache v2 software can be included in GPL v3 projects, whereas in contrast GPL v3 software cannot be included in Apache projects².

Furthermore, mixing OSS and proprietary code may lead to the obligation to make the proprietary code available as open source; this is known as the viral effect. In practice, there is a lot of “gray area” in designs that may or may not satisfy the requirements imposed by different OSS licenses. Ongoing work exists to provide proper definitions for this aspect [11], [12]. When integrating OSS components with new code, the restrictions and obligations which the licenses impose may depend on whether the new code is considered as derived (derivative) or combined (collective) [19]. Also the interpretation may depend on how the component is used: as a redistributable product, as a hosted service, as a development tool, or for internal use [20]. Finally, open source legality interpretations are subject to the way software is implemented, packaged, and deployed [10].

The relation between software architecture and licenses has been addressed in several prior research works such as [21]–[23]. Furthermore, as number of so-called license analysis techniques and tools have been proposed to identify the licenses (e.g. Ninka³) used and to verify license compliance in source code packages (e.g. FOSSology⁴, ASLA [24] and OSLC⁵). These techniques however are mostly useful in analyzing ready packaged software systems and their corresponding architecture but give little guidance, with respect to licensing issues, for software developers during the development activity itself. We formulate those guidelines as legality patterns.

III. PATTERNS IDENTIFIED AND PROPOSED STYLE

Based on these issues, we identify a set of patterns and then propose an integrated architectural style for addressing all of the above requirements.

Architectural patterns are not invented by researchers: they must be found repeatedly in practice. The characteristics of patterns lead to different behaviors and different responses, with associated costs and benefits, in response to environmental conditions. For this reason there will never be a complete list of patterns: patterns spontaneously emerge in reaction to environmental conditions and as long as those conditions change, new patterns will emerge. In this paper we will argue that a small set of patterns are highly influential in the creation of successful open source systems, where success is measured by evolutionary growth and widespread adoption. We begin by discussing the core-periphery pattern.

A. Core-periphery Architectural Pattern

This pattern employs a single, small, extensible core (which is typically highly coupled) surrounded by a much larger periphery (which typically exhibits low coupling). This pattern is intuitively suitable for meeting the requirement of ultra-widely distributed development. The core typically presents itself to the rest of the system as a “platform”, which the periphery accesses through a set of APIs [9], [16]. The APIs can be defined such that they are able to act as interfaces in terms of legal issues as well [10].

The core is developed by a community that is tightly coupled – a small set of project “committers”. Committers are developers who have achieved sufficient status in the project to be entrusted with its architecture and strategic direction. In general, tightly coupled components are difficult to understand and to modify. However, this complexity is tractable if the core is kept small (relative to the size of the entire system) and maintained by a small, committed team of project leaders with long experience and technical depth. The core makes few constraints on how the components at the periphery use it, or interact with each other. The periphery is relatively unconstrained; it builds on the services and capabilities provided by the core. This under-determination liberates stakeholders at the periphery to use the core, and to enhance the system in ways that have not been anticipated in the original design. This enables novel compositions and extensions of such systems. The core defines the basic quality attributes achievable by the system; its performance, security, availability, modifiability etc. These qualities are achieved based on the resources exposed by and controlled by the core – resources that it makes available to the periphery. The core also defines the most basic functionality of the system: for Apache’s httpd (web server) this functionality is responding to HTTP requests; for Linux this is managing system resources such as processors, memory, and I/O devices. The core provides “extension interfaces” which enable developers at the periphery to innovate while remaining largely unconstrained. In many cases the extensions may contain extensible cores of their own (e.g. Apache’s rich structure of extension modules).

The vast majority of what an end user sees and uses – the applications, user interfaces, devices, etc. – are actually part of the periphery, not a part of the core. In this respect, the core can even be seen as a platform or product line, whereas extensions can be seen as product specific components.

B. Legality Patterns

Given the potential combinations of licenses, the way OSS are allowed to interact with each other must often be considered on an application-specific basis. Figuring out all the details is time-consuming and tiresome. Therefore, once

²http://www.apache.org/licenses/GPL-compatibility.html
³http://ninka.turingmachine.org/
⁴http://www.fossology.org/
⁵http://www.sourceforge.net/projects/oslc
a design has been found adequate, its reuse can save a lot of time and effort.

Following the spirit of design patterns [6], the concept of legality patterns has been proposed as practical means for integrating OSS components [10]. The generic goal of a legality pattern is to define a name that extends our design vocabulary, a context and problem that define when the pattern is applicable, and a solution that defines how the problem is solved. In addition, known implementations and examples as well as special considerations are often used to demonstrate how the pattern is applied.

Legality patterns that address interaction and integration of OSS components are the most relevant and critical to our research [10], [19]. The core concept in all legality patterns is similar: to loosen the dependency of components on each other so they can be more easily replaced with other implementations if needed. Furthermore, by being independent of one another they will not be considered derivative work, and hence subject to licensing conditions.

Below we briefly describe five of the most important and common legality patterns, which are used in subsequent sections. A more comprehensive discussion of legality patterns, may be found in [10]. Note that this list of legality patterns has emerged from developers; they have not been tested in court as far as we know. Furthermore, it is obviously possible to explicitly define a license that will not enable the use of these patterns in some contexts.

The five patterns used in this paper are:

- **STANDARDIZED INTERFACE**: where a number of communicating components adopt a standardized way of communicating, as a means of loosening component-specific dependencies.
- **DYNAMIC LINKING**: where components are linked at runtime hence easing the substitution of independent, functionally similar components.
- **REPACKAGE**: where a piece of software is simply repackaged and re-released under a different license than the original one.
- **TIER**: where an intermediate tier is interposed between proprietary code and an open source component to remove direct dependencies (such as calls).
- **USER DELEGATION**: where a user is instructed to build a system by a specified set of steps.

C. Integrating the Core-Periphery Pattern with Legality Patterns

The core-periphery pattern describes a way to structure a complex software system as a set of core or kernel services surrounded by a periphery of application-specific software. The legality patterns describe how the core can be kept at arm’s length from the periphery (or how periphery components can be at arm’s length from each other) to avoid licensing problems. To address the architectural and licensing challenges simultaneously we propose an integrated architectural style, called CPL (Core Periphery Legality), in which legality patterns are used to define how a single, extensible core may be extended, with an option to use different licensing strategies.

Figure 1 depicts the structure of the CPL architectural style. Legality patterns are used to loosen the dependency of the periphery elements on the core. In contrast to other forms of pattern collections like composite patterns (e.g. the Bureaucracy pattern [25]), the CPL style is “open” in the sense that no specific pattern configuration is predetermined. Instead, the use of legality patterns is determined by the corresponding periphery element and the restrictions on connections it must make with the core.

![Figure 1. CPL Architectural Style](image)

IV. Empirical Study

Our hypothesis is that the adoption of the CPL architectural style will significantly increase the probability of successful OSS development. The proposed style can simultaneously support different types of licenses as well as both proprietary and OSS software. While we can not prove this hypothesis, we offer evidence in support of it by studying a set of well-known OSS, as both positive and negative examples. We note that this style has been used in all of the positive examples that we studied – which are among the most successful OSS projects of all time.

Several negative examples – of how the wrong choice of an OSS license, or lack of preparation for OSS licensing issues – have been previously noted by [13]. For example in 2001 OpenBSD was forced to re-implement its IPFilter package because it was directly depending on...
software with an incompatible license. An overview of the different licensing issues has been provided in [26]. The gpl-violations.org project\(^6\) lists a number of cases of GPL infringing organizations that led to court cases.

**A. Linux Kernel**

Linux distributions\(^7\)^\(^8\)^\(^9\) include many supporting programs; technically the “Linux operating system” refers only to the kernel. The architecture, based on a single extensible core, and the ability to extend this core, can be discerned at two hierarchical levels (see Figure 2). At the outermost level the core is entire the Linux kernel, and individual applications, libraries, resources and auxiliaries act as extensions to the Kernel’s functionality. Digging into the Linux Kernel however, we can once again discern a core-periphery pattern. Inside the Linux Kernel, modules are defined to enable parallel development of different subsystems. The functions that one expects to find in a Kernel are all present, but they are designed to be separate modules.

For instance, as shown in Figure 2, there are modules for processor/cache control, memory management, resource management, file system interfacing, networking stacks, device I/O, and so forth. All these interact, but are clearly separate modules within the Kernel. In addition to the modules constituting the Kernel, there are also components that add other functionality. For example device can be loaded and unloaded, often while the system is running. Furthermore, as is common in OSS, everyone can change or extend functionality by contributing patches. Patches that are favored by the community will eventually find their way to the core itself.

The licensing model of Linux is complex. The Linux Kernel is distributed under the full GNU General Public License (GPL): all components, modules, and extensions linked with the Kernel should be open source, and must be published only under licenses compatible with the GPL. In the case of Linux, however, the Kernel system call interface is an example of a well-documented and standardized interface. The Linux Kernel license includes the following clarification “The copyright does *not* cover user programs that use Kernel services by normal system calls – this is merely considered normal use of the Kernel, and does *not* fall under the heading of ‘derived work’”. The “normal system calls” of Linux embodies the concept of the STANDARDIZED INTERFACE pattern. Additionally, the Linux Kernel provides specific interfaces, which are licensed with the more liberal GNU Lesser General Public License (LGPL). This means that all components linked with modules that use the LGPL interfaces can be published under any compatible open source or proprietary license. Such intermediate modules implement the TIER pattern.

An example of this architectural pattern is the Linux STREAMS (LiS) module. LiS loads into the Kernel dynamically and uses the open Kernel interface to talk with the core directly [27]. This way the producers of proprietary drivers linked with LiS need not publish their sources. Finally, Linux is a distributed system that combines a number of internal and external components. Patches and extensions conflicting with the GPL can be integrated with the Kernel by having users follow specified procedures – an example of applying the USER DELEGATION pattern.

**B. Apache httpd**

The Apache HTTP Server Project (httpd for short) develops and maintains a secure, efficient and extensible HTTP server\(^10\). Apache httpd is widely used and extremely popular; since April 1996 it has been the most popular web server on the Internet.

The httpd architecture is organized in terms of modules (see Figure 3). The use of modules also affects the core of the system. One of the modules, referred to as the Core module, defines features that are always available. Variant features are implemented in separate modules. For example, multi-threading is implemented in separate modules, as well as environment-specific versions of the server. Additional modules exist to enable the use of languages for web-based applications, such as Python and PHP. This module-based

\(^6\)http://www.gpl-violations.org/
\(^7\)http://www.debian.org/
\(^8\)http://www.redhat.com/
\(^9\)http://www.suse.com/
\(^10\)http://httpd.apache.org/
design allows the webmaster to configure which features will be included in httpd by selecting which modules are loaded. The loading can take place either at compile or runtime. The modules are managed using a special tool called apxs. Since modules are a central concept in Apache httpd, distributions are built using them. The basic httpd distribution delivers a restricted set of modules; the most common ones. The user can extend the system with more application and context-specific modules.

C. Firefox Browser

Descended from the Mozilla Application Suite, Firefox is one of today’s most popular browsers with its 37% market share. One of the design goals of Firefox has been to enable 3rd party extensions, upgrades, and modifications. This added functionality can be implemented with two mechanisms – plugins and extensions (see Figure 4).

Firefox plugins are shared libraries used for displaying content types that are not natively supported by the browser. For instance, the Adobe Reader plugin for viewing PDF files inside the browser is ubiquitous. Firefox plugins are built using NPAPI\(^\text{11}\), a cross-browser interface for creating plugins, although older, deprecated plugin technologies still exist. Clearly, NPAPI can be regarded as an instance of the STANDARDIZED INTERFACE pattern. Firefox extensions are a mechanism for adding new functions, ranging from individual user interface buttons to entirely new features. The purpose is to allow customization without creating monolithic applications.

An extension is a collection of files and folders compressed into a single zip file with the .xpi (zippy) extension. Themes and language packs are also extensions but with a different internal file structure. Both plugins and extensions can be distributed in binary format without providing access to the associated source code. This makes it possible to use different licenses and even proprietary code when composing them [28]. In practice, plugin or extension specific End-User License Agreements (EULA) can be required before their installation can proceed. This is yet another example of the USER DELEGATION legality pattern. Firefox is provided under the Mozilla Public License (MPL): a weak copyleft license that bounds open source and proprietary code in one package licensed with MPL. Therefore, developers are not obliged to publish the source code of the entire product, saving their proprietary parts in a “black box”, building on the idea of the REPACKAGE pattern.

D. Discussion

The architectural evolution of complex open source software systems has been widely discussed. For example, the authors in [29] discuss a set of generic design and evolution principles in the Eclipse SDK. The assessment may often involve the reporting of different statistics related

\(^\text{11}\)https://developer.mozilla.org/en/Plugins
to design properties such as modularity and coupling [18]. This has been illustrated with the two cases of Linux kernel and Mozilla [17]. Our work adopts a different assessment technique. Our hypothesis was that the adoption of the CPL architectural style will significantly increase the probability of successful OSS development. Not only are each of the above examples highly successful, but each owes much of its success to its extensibility and scalability, which are directly supported by the core periphery pattern [9] and the legality patterns [10]. Examining the adoption and effects of our architectural style throughout the evolution of those projects will require data from the core development teams.

V. APPLYING CPL

As one purpose of an architectural style is to provide guidelines for new development, we conducted a case study to observe the applicability and advantages achieved by the use of CPL. The case study was done employing Collaborative Practice Research (CPR) [30] to gather feedback on the applicability of the proposed approach. We, the researchers, provide consulting support on the CPL style, making software reviews, and performing assessments of the design.

A. Case Description

SOLA (Solutions for Open Land Administration) is an ongoing OSS project supported by FAO (Food and Agriculture Organization). The goal of the project is to create an open source land registration and administration system that will be deployed in at least three pilot countries, Nepal, Ghana, and Samoa.

The design of SOLA has been driven by a number of non-technical requirements. The most important one is that SOLA is to be released as open source and that a community would be established to maintain it. There are two main phases: 1) a generic phase where the core components of the system are developed by a closed team, and 2) an application phase, where the system is adapted to the contexts of the three countries. Local teams will spearhead the adaption process. The outcome of phase 1 is a cohesive generic core that represents the essential functionality of a land administration system. This core must provide adaptation APIs and extension mechanisms. In phase 2 SOLA is augmented with peripheral elements specific to the context of pilot countries. There should be loose coupling between these elements and the core.

The second important requirement is that SOLA should be developed with a maximum use of existing OSS components. This would minimize development costs. A related factor has been to make it possible for proprietary companies to use modified versions of the software without the need to release back the modifications. It is essential to use a community-friendly open source license for to encourage contributions. However, extensive use of open source components should address the legal integrity of the system with regard to licensing issues. Furthermore, the choice of OSS and their configuration should not limit the use of the resulting system. Consequently, the design of SOLA provided a natural ground for applying the CPL architectural style. Our idea was to implement the generic software as a single extensible core and treat the pilot country customizations as peripheral elements. Furthermore, we wanted to consider legality patterns early in the design phase to avoid license compliance problems.

B. Experiences

Driven by the proposed approach, the architecture of SOLA has been defined to encompass the idea of having a single extensible core. This is illustrated in Figure 5. The core runs an application server that deploys system services. Example core services include a Spatial component for managing land parcels, Cadastre for handling land administration data, and Hibernate for data persistence. In addition, the core implements security related services such as auditing, authorization, and authentication. The SOLA core provides three extension points: a data extension point for supporting databases, a presentation extension point for porting to client platforms, and a services extension point for including extra services.

Figure 5. Architecture of SOLA

The architecture of SOLA was developed based on the notion of “Customized Service”, where customized workflows could be easily developed and plugged in to meet the needs of the pilot countries. System modularity was achieved, in part, by using Enterprise JavaBeans (EJB), where new EJB components can

---

12http://www.flossola.org/
be created providing different functionality and configured for use as a Service. The extension mechanisms also include usage of non-proprietary WS-* standards for interoperability with different clients (i.e. .NET) and a Business Rules Engine that makes it possible to change and customize application behavior without redeploying the entire system. The localization support was provided by using Java Locale and Java Resource Bundles, while date formats was were accommodated by using open source scripts for data transformation.

Since a major motivation of the SOLA project was to be open source, all major system elements were based on existing OSS. The “Data Layer” was realized by the PostgreSQL database with the GPL-licensed PostGIS extension that supports geographic objects. Services are operated by the open source GlassFish application server with the web service stack extension Metro. The Business Logic Engine was implemented with the Drools business rule management system (BRMS), which is distributed under the Apache license. SOLA uses other popular open source libraries. Table I depicts example open source components and their licenses.

![Table I](image.png)

Many components included in SOLA are maintained by distinct open source communities. For instance, PostGIS and GeoTools are supported by the OSGeo Foundation, while the PostgreSQL database has its own large community of contributors. Support from these communities is useful for further development and maintenance of the SOLA project.

One of the constraints in choosing open source components was that they would meet the legality requirements of their various licensing models. The architecture was designed by explicitly taking into consideration a number of open source legality concerns. The legal constraints have been addressed by using several legality patterns. First, we implemented the DYNAMIC LINKING pattern. Dynamic linking was preferred over static linking to achieve more independence among the communicating components. All libraries have been combined as separate JAR packages, which are loaded into the system at runtime. One example of dynamically linked libraries with contradicting licenses is the combination of Apache Licensed Barcode4J and LGPL’ed JasperReports. Second, we applied the USER DELEGATION legality pattern. The concept of Beans was selected as the main customization technology. It allowed end users to attach new (possibly proprietary) modules to the system without negative consequences for compatibility of the components. For instance, the Digital Archive service can be easily replaced with a proprietary service that supports an existing archive used by the third parties. Third, we adapted the REPACKAGE pattern. The modified BSD license was chosen as the main license of SOLA because it is compatible with all other internally used licenses. Another reason for choosing the modified BSD as the project License is to allow commercial companies to develop proprietary software on top of SOLA components.

By adopting legality patterns we resolved the licensing concerns associated with our open source components. At times, we addressed licensing issues differently. For instance, our original plan was to use a version of the iText library that used the AGPL license, which is not compatible with GPL, and using AGPL would have meant that all of SOLA must use AGPL as a main license. In the end it was easier to revert to an older version of the same library using a more liberal license, Mozilla Public License.

### C. Findings

As SOLA development is still ongoing, it is too early to assess its success in terms of how it grows. The growth of the community is dependent on many project factors beyond the system architecture. However, we were able to gather evidence about the benefits of the CPL style. To evaluate the success of adopting the proposed approach, we interviewed the senior software architect of SOLA and collected some qualitative evidence. Here are our findings:

1. The CPL architectural style approach provides guidelines for designing SOLA: The need for a modular core contributed to the decision to use a two dimensional layering approach. Traditional Responsibility-based layering (e.g. Presentation, Business Logic, Data) was used to segment the architecture into manageable blocks. Web services and EJBs were used to enhance the opportunity for reuse of the Business Logic layer as well as support extension via the implementation of new web services and EJBs. The Data Layer consists of a series of data schemas that limit direct data dependencies and simplify integration of SOLA components with other systems.

2. Legality patterns influenced the SOLA architecture during design and implementation: A consideration of legality issues restricted the range of technologies and components considered. In most cases the open source technologies are the equivalent of or better than non OSS options, however in 1 or 2 cases they had to enhance the base
OSS components to meet their needs. One example is the GeoTools library. GeoTools provides a comprehensive set of libraries and tools for the manipulation of spatial data but the GUI component accompanying the library is relatively basic. They enhanced the base GUI component to satisfy the requirements for the SOLA Map Viewer. However, they have deliberately avoided integrating with components using Strong Copyleft licenses so that they could maintain the BSD license for SOLA. In one case that meant using an earlier version of a particular library that was originally LGPL licensed, but had changed to GPL for more recent releases.

3) Other benefits for designers and developers: At first the CPL style introduced complexity. It took a while for the team to establish the appropriate implementation patterns to use within the broader architectural style, and some refactoring was required before they settled on the structure and functionality of the common components. However once that was established, the team was able to quickly extend the software with new features and functionality through the reuse of common components.

4) The CPL architectural style influenced the success of SOLA evolution: The architecture can be adapted to satisfy a broad range of possible needs and integration scenarios. Given that SOLA primarily targets government agencies, aspects such as security, reliability and compliance with common e-government guidelines have also contributed significantly to the choice of technologies used for SOLA.

Adopting the CPL architectural style required a learning period before successful adaption to SOLA. The main motivation of adopting and using the style has been to prepare SOLA for custom implementations and further evolution. The legality part of the style has been used as an awareness tool of open source licensing concerns.

VI. CONCLUSIONS

Architecture is a critical issue for enabling successful OSS development. As vast and varied as OSS are, they share peer-produced, crowdsourced system characteristics and they share a set of common legal issues. This paper has analyzed the requirements of OSS development, identified a set of architectural design patterns that are critical to successful large-scale OSS development, and proposed the CPL architectural style integrating the core-periphery pattern with legality patterns to address the architectural and legal concerns simultaneously. We argue that the two aspects must be addressed simultaneously to have positive design outcomes as they impact each other. We have validated the CPL style through three studies of successful OSS systems that have utilized different legality patterns to extend a common core. We then conducted a case study on a new OSS project, SOLA, to understand the effects of employing the CPL style. Our results showed that the CPL style has been useful in avoiding design minefields and in amalgamating the different components into a larger entity in a sustainable fashion.

Clearly, there are other issues for successful OSS system development such as quality of documentation, existence of good examples, mentoring, tutorials, crowd management, and leadership. But we believe that architectural support is foundational and without it the other techniques will be insufficient to support large-scale growth. To summarize our findings, the architecture of an OSS must support two important properties. One is the scalability of distributed peer-production, and the other is the ability to integrate systems with different license types, including both open source licenses and proprietary. Our empirical study shows that the most successful OSS projects embody these two architectural properties, realized by our proposed patterns. And our case study shows an existence proof that our proposed patterns are effective in structuring new development aimed at attracting a vigorous and self-sustaining community.

We do not claim that we are the originators of the architectural patterns presented in this – patterns are discovered by generalizing from a population of existing designs; they are not invented. The patterns we describe have been used many times, as evidenced in their frequent presence in highly successful real-world open source architectures. Our contribution is the identification and integration of the core-periphery pattern with legality patterns.

While OSS has been previously associated with patterns, this research has primarily focused on other perspectives, such as patterns of evolution [31]. The closest use of architectural patterns in the same sense as we have proposed is by Baldwin and Clark in [32], where they propose the “architecture of participation”, composed in terms of modules and option value. However, Baldwin and Clark focus on what this means from a developer perspective rather than the definition of architectural solutions that can be reused. Furthermore, the authors of [33] view open source platforms as a separation of core code from functional add-ons and user interface add-ons. This clearly resembles the core-periphery pattern.

REFERENCES


