Open Source Legality Patterns: Architectural Design Decisions Motivated by Legal Concerns

Imed Hammouda and Tommi Mikkonen
Tampere University of Technology
P.O. Box 553
FI-33101 Tampere, Finland
firstname.lastname@tut.fi

Ville Oksanen
Aalto University
P.O.Box 19210
FI-00760 Aalto, Finland
ville.oksanen@hut.fi

Ari Jaaksi
Nokia
P.O.Box 1000
FI-33721 Tampere, Finland
ari.jaaksi@nokia.com

ABSTRACT
Complications emerge when various open source software components, governed by different licenses, are used in the same software system. For various reasons, these licenses introduce different privileges and requirements on the use and distribution of composed code, and are therefore often fundamentally incompatible with each other when combined arbitrarily. Consequently the way the different components can be integrated requires attention at the level of software architecture. In this paper, we introduce open source legality patterns – architectural design decisions motivated by legal concerns associated with open source licensing issues and licenses themselves. Towards the end of the paper, we also review some related work and discuss why it is important to create common guidelines for designs that mix and match different open source systems and proprietary software, and provide directions for future work.

Categories and Subject Descriptors

General Terms
Management, Design, Legal Aspects.

Keywords
Design patterns, open source licensing, legal concerns.

1. INTRODUCTION
Software architecture has been standardized to refer to the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution [7]. In the design of software architectures, the concept of architecturally significant requirements is often high-lighted. Moreover, the architect’s role as a key character in orchestrating different stakeholder requirements into a single system has been widely recognized [10]. Obviously, not all of the requirements coming from stakeholders are architecturally relevant, but some of them usually constitute the “load-bearing walls” of the software system.

Commonly identified stakeholders of software architecture include testers, product managers, users, designers, marketing personnel, and so forth. Architecturally significant requirements resulting from such perspectives commonly include such quality attributes as testability, scalability, understandability, modularity, flexibility, and so on. We believe that there is a new emerging view to any software system that should be taken into account extensively: the legal view. To some extent, the legal view has been taken into account in designs as encryption and safety requirements. However, due to the increasing use of open source systems, where licensing issues differ from the conventional proprietary setting, a more holistic view towards legal issues associated with open source components is needed.

In order to simplify legal justifications regarding licensing issues, keeping different components with different licenses in isolation from one another is of crucial importance. Consequently architectural design of a system, which defines guidelines for modularity of the system, is affected by the different licenses, in addition to technical details which have commonly formed an important rationale for design. We believe that documenting such architecturally important design decisions is a necessary step towards more practical use of open source components in a business context. In analogy to design patterns – recurring design decisions in the presence of certain forces [5] – these decisions can be referred to as open source legality patterns, i.e., recurring design decisions taken due to legal forces traceable to open source licenses.

In this paper, we introduce the concept of open source legality patterns in detail, and introduce a number of open source legality patterns identified in real systems and expert interviews. For the most part, the patterns we address have been geared towards business goals, where circumventing certain properties of open source licenses have been a necessity. Note that we do not claim that the patterns we present would hold up under strict legal scrutiny. Furthermore, we recognize that many people in the open source community would view them as contrary to the accepted community norms. Thus, we do not recommend blind use of the patterns as such. Instead, we simply state that in practice software
designers or experts have proposed the corresponding design decisions as a precaution in connection with open source licenses, but do not pose any legal interpretation.

The rest of this paper is structured as follows. In Section 2 we discuss open source licensing and some licenses that are most commonly used. In addition, we reflect on licensing requirements to software architecture. In Section 3 we introduce the concept of a legality pattern in more detail. In addition, we also introduce the format we use for describing patterns. In Section 4, which forms the core of this paper, we introduce a list of legality patterns. In Section 5, we briefly review related work, and in Section 6, we provide some directions for future work. In Section 7, we draw some final conclusions.

2. OPEN SOURCE LICENSES AND SOFTWARE ARCHITECTURE

During recent years open source software development has been gaining a lot of foothold in numerous contexts. These contexts include different domains such as operating systems, middleware, development tools, and end-user applications, to name but a few. Moreover, people commonly amalgamate more complex systems out of open source software, where contributions from different communities and proprietary software are combined into a single integrated user experience, thus benefiting from the work of open source communities in a commercial setting.

Several views on open source and free software exist. For the purposes of this paper, we assume a pragmatic view: Any piece of software that is licensed following the open source definition’s requirement qualifies. The definition is available at http://www.opensource.org/docs/definition.php. The goal of this wide definition is to overlook discussions associated with the role of the developers, and the way in which the system has ended up being open source in the first place, and to place the emphasis on legal issues.

In general terms, open source licenses are special copyright licenses that give more rights to the licensee than the copyright law. They make source code available under terms that allow for modification and redistribution without a payment to the original author. Moreover, they may also introduce certain restrictions. Some common requirements are to preserve the name of the authors and the copyright statement within the code. At times, restrictions are such that they also affect the use of and redistribution of the code, for instance in the form of introducing obligations to make also modified versions available under the same terms as the original piece of software.

Numerous acknowledged open source software licenses exist, including, for example, the GNU General Public License, GPL (http://www.gnu.org/licenses/gpl.html), the Lesser GNU General Public License, LGPL (http://www.gnu.org/licenses/lgpl.html), the Apache license (http://commons.apache.org/license.html), the Massachusetts Institute of Technology license, MIT (http://www.opensource.org/licenses/mit-license.php), and the Berkeley Software Distribution license, BSD (http://www.opensource.org/licenses/bsd-license.php). As of March 2010 site http://www.opensource.org/licenses/ lists 66 different licenses – and terms of different licenses vary considerably. Moreover, some licenses have several versions, and there are subtle changes between different versions. In addition, the list is by no means complete, and new licenses can be introduced if so desired. For example, a new license can add some minor differences to an earlier one, thus generating a discrepancy between the licenses, or a completely new license can be introduced.

Since different licensing terms are in no way coordinated, they can be – and commonly are – conflicting in their terms and conditions. In fact, even the most common licenses are often incompatible with each other because of subtle differences, and software released under them cannot be combined arbitrarily. Consequently, the integration of components distributed under different licenses may lead to complications, since all license terms must be complied with. Given the potential number of combinations of open source licenses, the way open source systems are allowed to interact with each other must often be considered separately on an application-specific basis. Figuring out all the details is time-consuming and tiresome work. Moreover, there is a lot of “gray area” where there are no absolute answers to whether or not a certain design decision respects associated licenses or not.

As an example, one of the simplest licenses, the MIT license (http://www.opensource.org/licenses/mit-license.php) is listed in Figure 1.

Copyright (c) <year> <author>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Figure 1. The MIT License.

In contrast to the simplicity and straightforwardness of the MIT license, there are more complex licenses. For instance, the GPLv3 license given at http://www.gnu.org/licenses/gpl.html in print using the same format as is used in this paper takes 7 pages, and includes over 5000 characters – a typical length of a research paper. Moreover, the terms included in the license description are obviously more complex than those included in the MIT license. Furthermore, even more complex licenses can be easily found, where really precise and detailed terms are laid down, clearly sculpted by a large corporate legal department.
Different flavors of licenses introduce a fundamental problem of incompatibility. It is common that a license requires that licensing terms remain the same when a new version of the system has been created. For instance, the GPL license requires that modifications made to associated code are distributed under the terms of the GPL license. Similarly, some other licenses such as Apache require that the license is used in new versions of the system. Open Source Initiative has tried to tackle the license proliferation problem with a special committee that categorized their approved licenses to the following five different categories of licenses (http://www.opensource.org/proliferation-report):

- Licenses that are popular and widely used or with strong communities (total of 9 licenses).
- Special purpose licenses.
- Licenses that are redundant with more popular licenses.
- Non-reusable licenses.
- Other/Miscellaneous licenses.

Although it is possible to release the system under two different licenses, the licenses should hold at the same time in the example, which is impossible in the general case – licenses simply are too different. The only exception is copyright holders, who can release the same system under multiple terms, commonly referred to as multi-licensing. Consequently, if the terms of the two licenses, used in the creation of a single software system, are not immediately compatible, software distributed under them cannot be mixed freely by arbitrary developers – the resulting piece of code and its licensing simply cannot satisfy all the obligations at the same time.

In addition to the problem of actual incompatibility, the fact that licenses are becoming overly complex introduces another challenge. It is difficult to figure what the exact terms of a certain license are to begin with. Unlike with proprietary software, where one can usually consult the owner of the piece of software, there may be no single owner of a piece of open source software with whom to negotiate the details of the license.

A further problem that is commonly raised in cases where open source is mixed with company-specific code is related to the obligations associated with derivative works. Some open source licenses, which are referred to as strong copyleft, have a strong viral effect. In essence, the viral effect requires that the recipients of the modified software get the same rights as the modifier has – including full access to the source code. Moreover, the same freedoms must be preserved and forwarded in modified versions of software. The viral effect is often considered harmful by companies developing proprietary software that interacts with open source components. The rationale is that due to the viral effect associated with strong copyleft, the company risks either violating license terms of open source software or being forced to distribute the whole proprietary system under the same license.

There are several solutions to the above licensing problems. One solution is to only use software under licenses that allow freely mixing different components. This approach is sometimes applicable, provided that software with liberal enough licenses exists. Another way to deal with the problem is to use a commercial, negotiable license, assuming there is a single licensing author that owns the copyright with whom to communicate, as is the case in many multi-licensing systems. However, the approach is not commonly available, since the number of copyright holders can be tens of thousands in a large open source system that has been maintained by a community for a long time. Yet another way to deal with the problem is to manage licensing issues strictly at the level of projects, as proposed in [6]. This can be accomplished by either converging to a common license or by simply sending patches that can be integrated into the core software and distributed under the general license. Finally, the problem can also be solved at the level of software architecture by using an interface strong enough to separate different software components so that viral infection no longer takes place. We will follow this approach in the rest of this paper.

If it is known that there are several open source components and licenses will be used but their exact details are still open, it becomes more tempting to select an architecture where components are isolated from one another. For example, consider the development of a multimedia player, where the following requirement is given:

“It must be possible to use different codecs, implemented by 3rd party developers, and are available with different open source licenses as well as proprietary ones.”

When designing the system, the requirement could lead to an architectural design where each codec is an independent entity, with entities separated with a standard interface that enables interaction of software components, each with different licenses. The core multimedia player module then simply acts as a loader and execution manager for codecs. Constructing an architecture where this design philosophy is highlighted is thus crucial. Furthermore, preferably the design of the architecture is based on simple rules and principles that can be repeated for each integrated codec.

3. TOWARDS OPEN SOURCE LEGALITY PATTERNS

In this section, we give an overview of open source legality patterns, and introduce a system we use as a running example when listing some open source legality patterns.

3.1 Open Source Legality Patterns

In practice, it is not self-evident what kinds of systems can be built with open source software, and how code distributed under different licenses can interact. With increasing business interest in open source, also legal issues have become more important. For example http://www.opensourcelegal.org lists a number of cases where open source licensing issues are becoming a key competitive issue either in the form of use or in the form of hindering others. While in reality some misuse of open source software arguably display complete overlook of licensing issues, a number of cases appear to be in a gray zone, where different interpretations exist. In fact, to some extent it can even be questioned whether or not the whole concept of open source licenses is legally sound.

We believe that it would be beneficial for both open source communities as well as for the companies benefitting from open source components in building their proprietary products to define technical guidelines for acceptable interaction, similarly to Free Software Foundation’s Practical Guide to GPL Compliance (http://www.softwarefreedom.org/resources/2008/compliance-guide.html). This is especially important for cases where the used
licensing model is viral. The concept of a legality pattern introduced in this paper is aiming at defining different ways of interaction between proprietary and open source components, with different licenses. The goal is to create a library of mechanisms from which it would be possible to define both those enabled and disabled by an open source license.

Following the spirit of design patterns [5], we have chosen not to invent the mechanisms by ourselves but to identify solutions found in existing systems. The patterns included in this paper have been identified in interviews with numerous industry experts and open source specialists, as well as lawyers specializing in software. Notice, though, that we do not claim that these patterns will automatically enable the use of open source components as a part of any system, but simply that these kinds of decisions have been taken in practice by software engineers to enable the use of open source components.

The generic goal of design patterns is to define a recurring problem in a context, a solution to the problem, and documented consequences. In our case, the situation is somewhat simplified, and the format we are using contains the following elements:

- **Name.** The name will extend design vocabulary and ease discussing the pattern.
- **Context and Problem.** Defines a situation in which the pattern is applicable.
- **Solution.** Defines the solution according to which the problem can be solved.
- **Implementations.** Introduces some known implementation mechanisms that have been used for this purpose.
- **Special.** Some patterns lend themselves to some special considerations. For those, additional remarks will be given.
- **Example.** We use the system described below as a running example throughout the pattern descriptions.

Since we are still in a somewhat early phases of this work, it is possible that the format will be extended later on. For instance, a list of licenses supported by the pattern would be a welcome extension. However, since our goal is to simply document what we have witnessed in practice, we do not wish to speculate which licenses would fall in a similar category in this phase of our research.

### 3.2 Running Example

The system we will be using as a running example is a simple weather prediction system. The system has a layered architecture as illustrated in Figure 2, with associated licenses.

The weather prediction system is used to extrapolate future weather on the basis of archived data and newly collected data. The example system consists of six components: remote data source, data collection, data archiving, data processing, data presentation, and client. Each component has its own licensing terms.

The data collection system collects data from a remote data source which could as well be provided by third party services. The data archiving component stores past data for future computation. The data processing component accesses the data from archives and data collection systems in order to perform computation on the basis of certain requirements. The presentation layer in turns gets the processed data and delivers it to clients in a specific format. A client is any device with browser capability such as an internet browser or a mobile device.

![Figure 2. An Example Weather Prediction System.](image)

### 4. SOME COMMON OPEN SOURCE LEGALITY PATTERNS

In the following, we introduce a set of legality patterns. The set is by no means conclusive. Rather, this is seminal work towards a comprehensive catalog on what kinds of design decisions are adequate for different open source licenses.

#### 4.1 Interaction Legality Patterns

Interaction legality patterns are legality patterns that are most commonly applicable within the scope of a single system that is to be distributed to end users. The basic idea in all of these patterns is similar, to lose the dependency of components regarding each other. Consequently, the different components can be more easily replaced with other implementations if needed. This in turn leads to considering them independent from one another, and will not be derivative work of each other.

**Pattern: Standardized interface**

*Context and problem*

A software component A links to a strong copyleft licensed (such as GPL) component B using non-standardized interfaces (hacks). For this reason, the source code of the component A should be released as it strictly ties itself to the source code of component B. Still, the developers would not like to release the source code of component A.
Solution

Use standardized interface calls to a strong copyleft licensed component to loosen the ties and dependencies among the two communicating components. In this way, component A is not tied any more to component B but can be integrated to any other product which offer same standard interfaces. In the case of Linux, 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’”.

Implementations

Embedded systems that can run on multiple platforms (such as Linux and Windows), software standard implementations, and different kinds of wrappers.

Special considerations

Developers should be aware of system interfaces to distinguish between standard and non-standard ones.

Example

Suppose the presentation layer in the weather prediction system links to the data processing component using a non-standardized data fetching interface. The presentation layer is a proprietary component, but the data processing component is an open source component, licensed under the GPL license. In this scenario, the source code of the presentation layer should be released as it strictly ties itself to the source code of the data processing component. A possible remedy would be to refactor all the non-standardized references to data processing component to relax the ties and dependencies among the two communicating components.

Pattern: Dynamic linking

Context and problem

A system is composed of a number of proprietary and open source components that are statically linked. If a proprietary component statically links to a strong copyleft licensed component, the source code of the proprietary component needs to be published under the same license terms of the open source component.

Solution

Switch from static linking to dynamic linking in the implementation. Dynamic linking infers “more” independence among the linked components [11].

Implementations

COTS, different plugin systems, dynamic linking facilities in operating systems.

Special considerations

While switching to dynamic linking, the developers should not tie the system to a particular strong copyleft licensed component, but leave the freedom for users to integrate with any components.

Dynamic linking pattern is in practice often used in connection with Standardized interface pattern.

Example

In the weather prediction system, presentation layer is a proprietary component which is statically linked to a strong copyleft data processing component (GPL). In this scenario, the source code of presentation layer needs to be published under terms of the GPL license. In order not to make the presentation layer subject to the terms of the GPL license, the data processing component should be loaded into the application at runtime. In practice this means that the data processing component could be replaced with another library that provides a similar interface.

Pattern: Data-driven communication

Context and problem

A system is composed of a number of proprietary and open source components with conflicting licenses. The components link to each other control-driven communication.

Solution

Avoid conflicting licenses by migrating from control-driven communication to data-driven component relationships.

Implementations

Systems that can be implemented around architectural styles such as Pipes and Filters architectural style, and blackboard. Piping in Unix type of environment.

Special considerations

Migrating to data-centric communication may come at the cost of overall system quality attributes such as performance, since interpreting the data may lead in increased processing requirements.

Example

The weather prediction system is composed of different open source and proprietary components. Suppose that the application has been developed in such a way that the communication between the presentation layer and data processing components is through control messages for data request. Assuming that the presentation layer is a proprietary component and that the data processing component is GPL'ed, the control-based inter-component communication leads to a license conflict. One way to resolve the conflict is to change the mode of communication from control to data messages. For instance, the data processing component could write data into a buffer, which is then read by the presentation layer component.

Pattern: Evaluator

Context and problem

The main program is only available as open source, and it needs the results of another program, which in turn is derivative of another open source system. Licenses of used programs are incompatible.

Solution

The main program initializes the execution of the auxiliary program with the desired set of data. Once the execution is
complete, the auxiliary program delivers the results to the main program.

**Implementations**

Many virtual machine based systems, in particular those that can be extended with scripts. Proprietary graphics accelerators that can be accessed from open source systems.

**Special considerations**

At times it may be unclear how the input given to the evaluator should be licensed, under program (e.g. GPL) or content creation (e.g. Creative Commons) licenses.

**Example**

In the weather prediction system, the data collection component needs the data from the remote data source component. The remote data source is also a derivative of another open source component. As a result, the licenses of the data collection and the remote data source components are incompatible. In order to resolve the problem we can build an auxiliary component. The data collection component initializes the execution of the auxiliary component with the desired set of data. After the completion of the execution of the auxiliary component, the results will be delivered to the data collection component.

### 4.2 Isolation Legality Patterns

Isolation legality patterns are commonly aiming at isolating open source components in a way that they would remain in the use of a single authority. This isolation in turn can liberate the developer from the responsibility to offer the upgraded version of the system for download. Most commonly such designs are used in web-based systems, but also other kinds of implementations exist.

**Pattern: Proprietary Server**

**Context and problem**

A system is composed of a number of open source components but the developers wish that certain parts of the system remain proprietary for business reasons.

**Solution**

Introduce a single server system and run all business critical code in the server.

**Implementations**

Numerous web systems.

**Special considerations**

What makes this solution possible is that a software package can be distributed using a license that is different from the one(s) used for the individual source files of that package.

**Pattern: Service interaction**

**Context and problem**

A system which includes user interaction is a derived work of a system-as-a-service (SAAS) like licensed (such as AGPL) software. User interaction requires that the source code of the derived work needs to be published. The developers would not like to release the source code.

**Solution**

Avoid user interaction by including an additional layer between the user and the open source licensed service. Since the consumer of the open source licensed software is a service and not a user, the requirement of publishing the code does not hold.

**Implementations**

Numerous web systems. Software as a service systems.

**Special considerations**

The GUI interfaces are moved from the open source licensed service to the additional layer. The user interacts only with the additional layer.

**Example**

Suppose, in the weather prediction system, the remote data source is under AGPL and acting as a SAAS. Since the weather prediction is a web based system and due to user interaction the source code of the derived work of the remote data source needs to be published. On the other hand, the developers are not interested to release the source code. In order to resolve this issue we can include an additional layer on top of SAAS that is the remote data source component. Since the consumer of the remote data source is a service the requirement of publishing the source code does not hold.

### 4.3 Licensing Legality Patterns

Licensing legality patterns concern the way the different software components should be licensed. In the following, we introduce 3 patterns geared towards such issues.

**Pattern: Repackage**

**Context and problem**

Some open source licenses such as BSD are so liberal that a derived work of corresponding source code can be published under a different license. However, relicensing is often considered as a risky and bad practice. Still, the developers wish to relicense the component to address the legal concerns of clients.

**Solution**

Repackage the derived source code and relicense the package. The source code itself keeps the original open source license. The derived source can be distributed under a new license.

**Implementations**

COTS.

**Special considerations**

What makes this solution possible is that a software package can be distributed using a license that is different from the one(s) used for the individual source files of that package.
Example

Suppose the data archiving component in the weather prediction system is under BSD license which is considered as a liberal license. According to the terms of BSD license, derived work of the data archiving component can be released under a different license. On the other hand, relicensing is considered as an uncertain practice. In this case, the developer may repackage the derived work of data archiving component and relicense the package. The data archiving component itself keeps the BSD license. The derived work can be published under a new license.

Pattern: Tier

Context and problem
A piece of proprietary code or code under an incompatible open source license wants to use an open source component with strong copyleft.

Solution
Add an intermediate tier layer that can be released under a license that is compatible with the copyleft license, but is less sticky so that a piece of code using its interface is not affected.

Implementations
COTS. Numerous interfacing and wrapper components and frameworks.

Special considerations
May lead to inventing new licenses that enable the intended use of the component, depending on what kind of a license is being wrapped.

Example
In the weather prediction system, presentation layer is a proprietary component which uses a strong copyleft data processing component (GPL). This would require releasing source code of the presentation layer component. A possible workaround is to add an intermediate tier between presentation layer and data processing that is released under a less stickier license such as LGPL or BSD. The intermediate layer would act as a business delegate to the data processing component.

Pattern: User delegation

Context and problem
A system is constructed of a number of open source components that are not compatible with each other due to the use of different licenses. Consequently they cannot be made readily available as a single package that could be downloaded, compiled, and installed.

Solution
Instead of making the system available as such, the user is instructed to follow a number of steps that are necessary for building the system. There are no compatibility concerns in the case an end user integrates together different open source components.

Implementations
Numerous patch systems.

Special considerations
If the user is instructed to build a system that potentially violates licensing terms, the case should be made explicit in the associated documentation. Also, the user is not supposed to distribute further the composed system. Otherwise, the same compatibility issues will apply.

Example
In the example system the data processing, data archiving, data collection and remote data source components are open source. Their corresponding licenses could be conflicting. Furthermore, the application can be used in different scenarios. Depending on the usage scenario, certain components may not be needed. For instance, the application could be about browsing historical data only. In this case, the data collection and remote data source components are not needed. In some other usage scenario, data archives might not be needed. Rather than providing the components as a one integrated system, which would mean that the company needs to address possible license conflict problems between the components, the system could be provided as a pool of plugins. It is then up to the user to compose the application out of a selected set of plugins.

5. RELATED WORK

Different kinds of patterns (like architectural patterns [2], design patterns [5], analysis patterns [4], etc) have become a central part of contemporary software engineering. Patterns document proven design experience. On the other hand, anti-patterns [1] have been proposed to describe design solutions that lead to negative consequences. Legality patterns take the role of both patterns and anti-patterns in the sense that they both describe an illegal situation and provide an architectural workaround to reduce the legal risks associated with the original situation. It was possible to organize legality patterns into different categories. However, unlike design patterns for example, it is not straightforward to organize legality patterns into a pattern language or system.

The research that comes closest to our work is license integration patterns [6], where the authors present a set of possible ways to integrate software originating from different open source projects, associated with different open source licenses. The paper goes on to study what means developers have in terms of licensing issues, such as converging towards a license that is most restrictive, or compose the system simply by submitting patches. As the goal is to show how the licenses integrate, not how to compose software where licenses need not integrate, the paper overlooks architectural issues we have addressed in this paper.

From a tool support perspective, the legality challenge of open source software has been partly addressed using so-called license analysis tools such as FOSSology [3], OSLC [8], and ASLA [12]. These tools provide functionality to identify the licenses used and to verify license compliance in source code packages. An extensive comparative study of these tools is presented from functional and quality points of view in [12]. Similarly, tool support for legality patterns could be introduced to validate architectural design against the legal requirements of open source licenses. Such tool support could as well benefit from the considerable improvement of tool support for patterns in recent Integrated Development Environment (IDE) systems and CASE tools (e.g. [9]).
6. FUTURE WORK
To begin with, one of the fundamental goals of this paper is to emphasize that guidelines are needed for using open source components in different contexts. We believe that the practices arising from the industry form a reasonable starting point for defining such guidelines. However, the view of license holders is of course of crucial importance, and therefore their view on how the systems licensed under different terms can be used must be made explicit. Open source legality patterns offer a practical way to define this aspect in terms of technology rather than legal terms. Consequently developers would benefit from listing the different ways a certain open source system can be used.

The list of patterns given in this paper is by no means conclusive, and we believe that there are a lot of open source legality patterns that are being applied, which have not been included in this paper. Identifying more patterns is of course an interesting direction for future work. In fact, even a pattern language based on legality patterns could be introduced to ease the application of the patterns. Similarly, the categorization used in this paper is by no means conclusive. It has been defined to be able to address the patterns we have documented.

Finally, developing tool support that helps in using open source legality patterns is also an option for further work. The tool could also support analyzing the use of different licenses that have been used in a system, as well as evaluate whether or not the used legality patterns really isolate open source components in the sense of legal view.

7. CONCLUSIONS
The way open source components can interact with each other and proprietary software has become an important architectural concern. In this paper, we have proposed that patterns are used for managing the way different components can interact to ensure that all licenses of open source components are complied with. We listed a number of open source legality patterns that we have observed in practice or identified in expert interviews. However, we do not claim that the patterns would hold legal scrutiny – we simply state that the designers’ intention has been to accommodate to licensing restrictions by using them. Consequently, we hope that such patterns would emerge as a practical tool that would enable license holders to define in technical terms what kind of use of an open source system is encouraged in connection with proprietary software and other open source components, with potentially conflicting licenses.

8. REFERENCES