Software Architecture Patterns for System Administration Support

ROLAND BIJVANK, HU University of Applied Sciences, Utrecht, the Netherlands
WIEBE WIERSEMA, HU University of Applied Sciences, Utrecht, the Netherlands
CHRISTIAN KÖPPE, HAN University of Applied Sciences, Arnhem, the Netherlands

Many quality aspects of software systems are addressed in the existing literature on software architecture patterns. But the aspect of system administration seems to be a bit overlooked, even though it is an important aspect too.

In this work we present three software architecture patterns that, when applied by software architects, support the work of system administrators: PROVIDE AN ADMINISTRATION API, SINGLE FILE LOCATION, and CENTRALIZED SYSTEM LOGGING. PROVIDE AN ADMINISTRATION API should solve problems encountered when trying to automate administration tasks. The SINGLE FILE LOCATION pattern should help system administrators to find the files of an application in one (hierarchical) place. CENTRALIZED SYSTEM LOGGING is useful to prevent coming up with several logging formats and locations.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures—*(Design) Patterns*

General Terms: Software Architecture, System Administration

Additional Key Words and Phrases: Software Architecture, System Administration

ACM Reference Format:

Bijvank, R., Wiersema, W., Köppe, C. 2013. Software Architecture Patterns for System Administration Support – L(October 2013), 14 pages.

1. INTRODUCTION

In the article *A plea from sysadmins to software vendors: 10 Do's and Don'ts* by Thomas Limoncelli [Limoncelli 2011], system administrators—the people responsible for installation, operation and upkeep of computer systems in a company—collected a basic list of do's and don'ts for software vendors in order to make the life of the system administrators more easy. The problems that are described in the aforementioned article have also been experienced within daily system administration practice, as interviews taken with system administrators show.

Among the suggestions given in the article to solve such problems are:

—DON'T make the administrative interface a GUI.

—DO create an API so the system can be remotely administered.

—DO use the built-in system logging mechanism (Unix syslog or Windows Event Logs).

—DON'T scribble all over the disk.

Authors' addresses: roland.bijvank@hu.nl; wiebe.wiersema@hu.nl; christian.koppe@han.nl

For the points in this list there is obviously a high agreement between system administrators on what the best practices should be. However, as system administrators are on the "receiving" end for a new or modified application, it is necessary to influence other parties who have a key position in creating or changing an application.

A role that fits this key position is the software architect. Among other concerns the software architect is responsible for the software architecture. The software architecture is the main design document for the software of an application and the design decisions taken in that document have a profound impact on the workload of the system administrators.

Patterns have been proven successful in imcreasing the quality of software architectures [Harrison 2011] and many patterns have been described, e.g. in [Buschmann et al. 1996; Buschmann et al. 2007; Fowler 2002]. As the focus of Software Architecture is often on realizing quality attributes, many patterns address them explicitly. Examples of these quality attributes as described in ISO 25010 are: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability. Examples of publications that focus on patterns for specific quality aspects are patterns for fault tolerant systems [Hanmer 2007] or security patterns [Schumacher et al. 2005].

There is one quality attribute where attention has been paid to mainly on a high level with respect to patterns: Portability and its sub-qualities Adaptability, Installability, and Replaceability. Patterns that address this issue in a more generic way are LAYERS [Buschmann et al. 1996] or ADAPTER [Gamma et al. 1994]. A number of concerns from system administrators are covered by the aforementioned attributes and patterns, but the mapping of the concerns on the attributes is not intuitive.

Both approaches — software architecture patterns for realizing the above described quality attributes and patterns that support the work of system administrators — don't touch some important aspects of the intersection of software architecture and system administration. Therefore we want to introduce a set of patterns which bridges this gap, based on the needs of the system administrators. This paper aims at influencing the software architects and the software architecture by providing patterns for software architecture that are endorsed by system administrators. We want to give ideas to software architects and application developers on how to improve their applications from a system administration viewpoint.

2. THE PATTERNS

There have been several initiatives to describe patterns from the perspective of a system administrator, but these are mainly focused on infrastructure and middleware. Examples of these initiatives are:

—Daniel Jumelet: Open Infrastructure Architecture repository (OIAr) [OIAm 2014] - this site provides a wide variety of infrastructure patterns for several working areas: Client Realm, Middleware, Network, Security + Support, Server, Storage. Beside this repository also contains architecture & design guidelines in the form of construction models at various levels and from various angles. It is constructed by making use of one of the most important tools of OIAm: The Building Blocks Model.

—Gregor Hohpe and Bobby Woolf: Enterprise Integration Patterns[Hohpe and Woolf 2003] - this book provides a consistent vocabulary and visual notation to describe large-scale integration solutions across many implementation technologies.

In this paper we present three software architecture patterns for system administration support:

—ADMINISTRATION API

—SINGLE FILE LOCATION AND STRUCTURE

—CENTRALIZED SYSTEM LOGGING

The patterns use an adapted version of the Alexandrian pattern format, as described in [Alexander et al. 1977]. The first part of each pattern is a short description of the context, followed by three diamonds. In the second part, the problem (in bold) and the forces are described, followed by another three diamonds. The third part offers the solution (again in bold), consequences of the pattern application — which are part of the resulting context — and a discussion of possible implementations. In the final part of each pattern, shown in *italics*, we discuss related patterns, offer a rationale for the pattern based on literature and present know uses.

ADMINISTRATION API

Context:
Software systems that are deployed in a professional environment often need to be maintained by a system administrator, e.g. accounting software or software in a company that is used for managing customer relationships. These administration tasks include adding or removing users, updating the system to a new version, or establish and maintain connections to other third party software. Most applications provide an administrative interface for system administrators to perform these tasks.

❖ ❖ ❖

Problem and forces:
If the administrative interface is a GUI, many of the standard administration tasks can not be easily automated. Repetitive tasks have to be manually completed again and again, which leads to a high frustration of the administrators. It also can be hard to get remote access to such a GUI.

Unexpected Automation Demand. Good system administrators have their own ways of organizing their administration tasks. They strive to automate many parts, often in unexpected ways, and a GUI minimizes the possibilities of doing so.

Platform Diversity. The operating systems—e.g. Windows, Unix, or MacOS—which administrators are using for their administration tasks frequently differ from the OS the application to be administered is running on.

Rise of the Cloud. The lower cost to deploy systems in the Cloud leads to more systems being deployed and subsequently to a higher workload for the system administrators if they do not adopt more efficient means for system administration.

Increasing Rate of Upgrades and Deploys. The Agile and DevOps development lifecycles where software upgrades are deployed on a weekly or even a daily basis, as opposed to the quarterly and yearly deploy cycles of more traditional software development methods, imposes tight control, predictability and efficiency on the deploy, installation and configuration of software [Humble and Farley 2010].

❖ ❖ ❖

Therefore: Provide an API for all required administration functionality. Make this API externally available, easily accessible—e.g. with a serializable or command line interface—and well documented, so that admins can automate administrative tasks and integrate it easily in the administration processes.

Solution description:
Offering an administration API provides much more flexibility for the system administrators to administer the systems in the way they think fits best. It gives them enough freedom to integrate the administration in existing processes. In order to be able to offer this high degree of freedom regarding the usage of the API, the system developers have to carefully design it and to offer the administration functionality in appropriate abstraction levels. This means that the API should be fine-grained enough.

The tasks of the system administrators are quite wide e.g.: installation, maintenance, scheduling repair, performance monitoring, backup & recovery, defining and maintaining usage and security policies etc. For most of these tasks an API can increase the efficiency and quality of the administration processes.

Automating administrative tasks reduces the number of errors that normally occur in manual execution such as the omission of steps or typing errors in commands. In a script such errors will also occur while programming the script, but once discovered can be fixed for subsequent usage of the script.

Tools for automation can make use of the administration functionality if they can connect to the provided API. For example, the right API helps to automate tasks that are part of a new employee account creation process. [Limoncelli 2011].

To securely expose administrative features utilize a PROXY [Buschmann et al. 1996]. The PROXY can include an authentication and authorization mechanism and block all unauthorized access attempts. This will be discussed in more detail in the implementation description below.

If the system evolves, the API is also likely to change which might require adaptations the system developers are not aware of. This is a general problem in interface- and component-based development and needs to be addressed in the design of the API too.

Providing an API might require more elaborate documentation compared to a more intuitive and self-explaining administrative GUI. The allocation of user roles using such API might e.g. require the correct spelling of the user role names. A misspelling of these names can lead to errors or have other unwanted effects. A GUI can offer a selection list including all user roles and possibly an extra explanation of these roles in a separate window section. This minimizes the need for extra documentation. The API should therefore include an extensive help, containing all information necessary for using the provided administration functionality. For the same reason the API should include a good exception handling in combination with clear error messages.

In the most simple cases the pattern is a specific variant of a SERVICE LAYER [Fowler 2002]. In this case it does not contain any logic, but simply forwards all requests to already existing configuration subsystems that offer the administration functionality. How these internal configuration subsystems are implemented is hidden to the user of the administration API. They can use this administration API using their own administration tools, e.g. a GUI or automated scripts. This is shown in Figure 1.

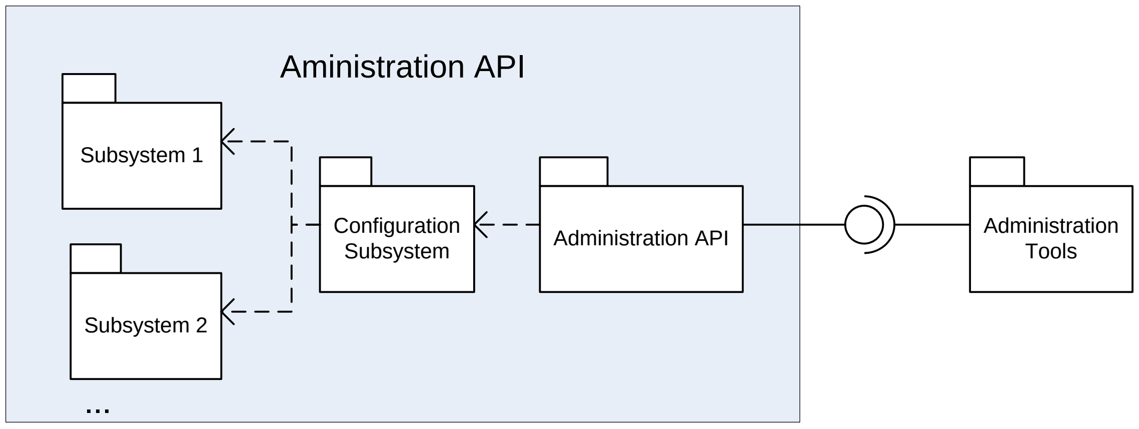

Fig. 1. Main solution structure of PROVIDE AN ADMINISTRATION API

If the administration API should not be publicly available due to security reasons, a PROXY [Buschmann et al. 1996] could be used to adequately address this issue. Figure 2 shows the main design. The protection proxy needs to include some mechanism for authentication and authorization of the requester.

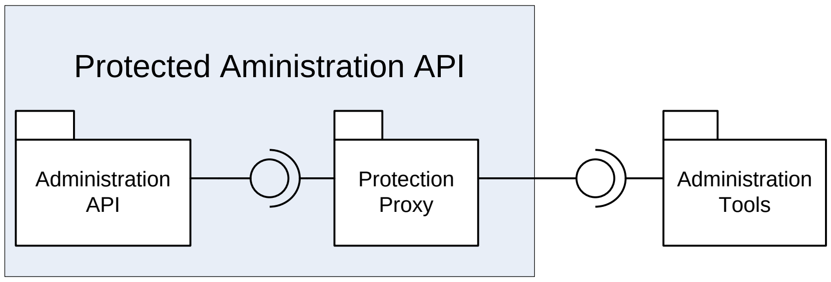

Fig. 2. An administration API including a protection proxy for security reasons

In certain cases the implementation language of the system and that of the administration API are different. Main reason for this could be that the administration API is required to be provided in a specific scripting language that suits the administrators' tasks best. In that case the administration API subsystem also becomes a specific kind of an ADAPTER [Gamma et al. 1994] between these two implementation languages.

❖ ❖ ❖

Rationale:
The problem of different platforms used for the system and in the administration environment can be minimized by making use of cross-platform scripting languages like Python, Ruby or TCL. This is also a certain advantage above graphical administration interfaces, as it removes the platform-specific issues caused by the GUI technologies. In combination with such a cross-platform scripting language this pattern shows its real strength as one can uniformly approach the administration API on any given platform.

Ideally, any changes in the system itself do not lead to changes in the administration API. However, if also functionality of the system regarding its configuration is changing, then also the API likely needs to be changed. The tools of the administrators are dependent on the API both syntactically and semantically in varying degrees. Unfortunately are both dependency types interrelated: the less syntactic the dependency is, the higher it is semantically and vice versa. One criterion that can be used for determining if the API should decrease the syntactic or the semantic dependencies is how easy it is to adapt the connection to the API on either syntactic and semantic level. If the interfaces are easy to adapt on both sides, then one should prefer more syntactically dependent interfaces that explicitly contain the semantic information in the naming of the methods and parameters. If the interfaces are not easy to adapt, then the syntactical dependencies should be low by using more generic interfaces that merely require different parameter contents but no interface adaptations.

Known uses:
Microsoft Sharepoint 2013 provides a rich set of APIs. Especially the server object model API offers all required functionality for administration tasks as "backup, farm health and diagnostics, logging, farm and web application management, upgrade, deployment, caching, and Windows PowerShell customization." [Microsoft 2013]

Another known use of this pattern can be found in Software-Defined Networking. This networking architecture is designed to use standardized APIs for defining and reconfiguring the way data and resources are handled within a network and to make interfacing and reconfiguring the network and its components easier [Kirkpatrick 2013].

One possibility of implementing this administrative API in the Java programming language are Java Management Extensions (JMX) [Oracle 2014].

Google also provides administration APIs for managing Google Apps, e.g. an API for monitoring the activities of account's administrators or for version conflict detection [Google Apps Platform 2014].

SINGLE FILE LOCATION AND STRUCTURE

Context:
Files are an established mechanism used by applications to store and retrieve configuration, libraries, state, data etc. Newly developed applications tend to use files in their own unique manner and store files in various locations. This may lead to having files that are dispersed over different folders or hidden in system-folders of the Operating System. System administrators want to be able to perform version control on the files.

❖ ❖ ❖

Problem and forces:
Having dispersed files causes system administrators to have difficulty in finding the files necessary for their tasks during the life cycle of an application.

—Distributed Applications.
Many applications consist of different subsystems, which often require subsystem-specific administration tasks. These subsystems are in many cases developed by different teams, resulting in dispersed groups of similar artifacts for each subsystem. This situation is well suited for developers as they can work in parallel. During deploy or system administration activities this can be a burden because of the way they have to look in different locations.

—Hard-coded Locations.
It happens often that developers put the location of the configuration files in source code and provide no parameters or interface to influence this location. This means the path can only be changed by building and deploying a new version of the application. Running multiple instances of a program on a machine with different parameters is effectively blocked by this approach. Additionally it can pose security risks if the file location is in a privileged location such as `C:\Program Files` for Windows based systems.

—Pollution.
When a file of a module isn't used anymore it will easily remain in disuse and get overlooked which causes pollution of your hard disk.

❖ ❖ ❖

Therefore: Put all related files in a folder hierarchy relative to one location on a file system or repository. Make the path of this location configurable.

Solution description:
Analyze the files and folders of the application, group the files that logically belong together and should be at the same location in a folder e.g.: the binaries of a system, the configuration files and the data files. In the case of log files one should first consider to use CENTRALIZED SYSTEM LOGGING.

Ideally it should be a structure that is re-used across applications (see figure 3) that are installed on the same server. This provides consistency for the system administrator, but also might help to overcome possible redundancies of files (e.g. keeping track of the language used). It furthermore serves as a clear guideline for the developers and could also be included in a reference architecture.

For reading the contents of configuration files PROPERTY LOADER and related patterns [Wellhausen et al. 2010] can be used.

If an application is deployed several times on a server it may be necessary to include the name of each specific application instances in the path on the file system. E.g. use `/somewhere/theapp/internal` and `/somewhere/theapp/external` if an application instance is deployed once for internal users and once for external users on the same server. This will separate the files for each application deployment. A common practice in these cases is to a symbolic link to share common files across all deployed instances.

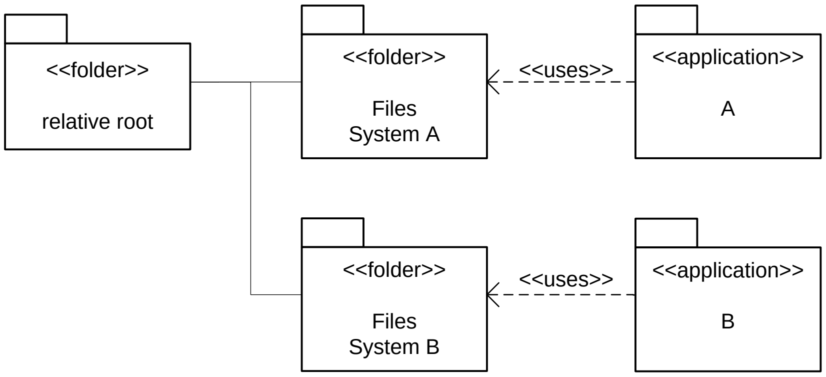

Fig. 3. The hierarchical file structure

The applications that do file access should not use the native File IO Libraries but should use a FACADE [Gamma et al. 1994] for accessing files (see figure 4). This FACADE provides the basic file IO functionality and prohibits absolute path access. The FACADE is using a configurable absolute path that is the root of all file access. The relative paths branch from that root path. This is best enforced in combination with a build server that checks which libraries are used from source code. The build should break when native File IO Libraries are used instead of the library that provides the FACADE.

If the folder structure to be used for applications is standardized across a development group, developers find it easier to navigate across an application and find the right files and folders. This can also be supported by providing skeleton solution or projects structures within the development environments such as Eclipse or Microsoft Visual Studio.

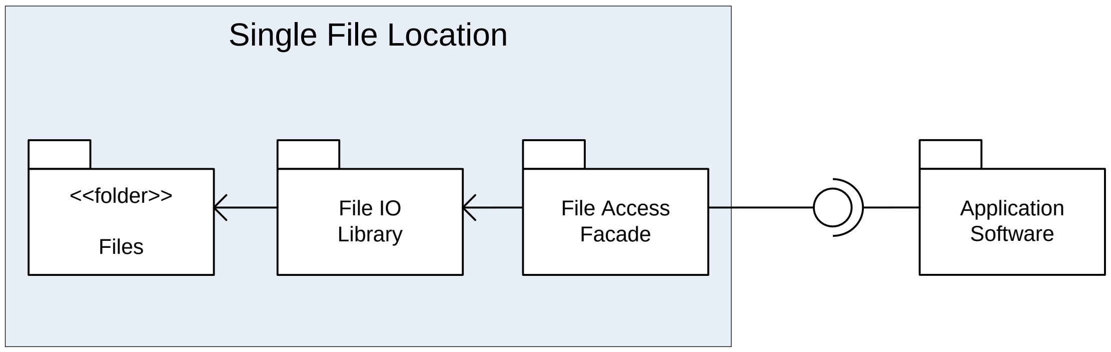

Fig. 4. Main solution structure for file access using SINGLE FILE LOCATION

❖ ❖ ❖

Rationale:
When not using this pattern the files of applications will be dispersed over several distinct locations which makes it hard to maintain the application.

Using SINGLE FILE LOCATION AND STRUCTURE is also more secure as this blocks access to other parts of the filesystem on the server as the implemented FACADE blocks access outside the root location. It is somewhat similar to a jailshell which is widely used to constrain linux users within their homefolder.

A nice example of the structure of SINGLE FILE LOCATION AND STRUCTURE without a FACADE *is for instance found in the way applications based on the Ruby on Rails (RoR) framework [Rails 2014] are structured. Every project starts with a pre-defined folder and file structure. RoR organizes all models, view and controller related logic in model, view and controller folders. Alle configuration settings are stored in a configuration folder and the Development, Test and Production stages related settings are stored in subfolder of the configuration folder. The framework provide internal relative paths to these folders so application can be stored in any location as long as the structure within the application folder remains the same.*

Another example is the Filesystem Hierarchy Standard [FHS 2014]. FHS defines the directory structure and directory contents in Unix and Unix-like operating systems, maintained by the Linux Foundation. Most Linux distributions follow the Filesystem Hierarchy Standard and maintain FHS compliance. Similar constructs are found in how OSX organizes applications and application data in file hierarchy

CENTRALIZED SYSTEM LOGGING

Context:
The application needs to provide the ability of logging certain events or actions for the application developer. The application developer is one target audience for the need of logging, the other target audience is the system administrator for the purpose of system management, especially monitoring of system (landscapes).

❖ ❖ ❖

Problem and forces:
Having a variety of logging formats and log-file locations makes it hard to monitor the state of a whole enterprise, including all running applications. In the case of an error it is hard to find the cause for it when multiple logs are dispersed over hundreds of servers without the right tools. Some problems which could be met when trying to integrate the wide variety of log files and their formats are:

Format Variety. A high variety of logging formats increases the complexity of integrating the information held within those several log files. It becomes a burden to nullify the different lay-outs of these log files.

Location Variety. When having a variety of log file locations the dispersion of those locations makes it hard to find those log files.

Information Granularity. Not only the formats might be varying, but also the granularity of information. This makes it hard to monitor all applications in a consistent way or to integrate the information in a consistent way for other statistical purposes like e.g. root cause analysis [Paschke and Schnappinger-Gerull 2006].

❖ ❖ ❖

**Therefore: Use the built-in system logging mechanism whenever possible (preferred solution).
If it is not possible, then define a standard format to be used by all systems and implement your own logger (alternative solution).**

Solution description:
First the preferred solution will be looked at. Many monitoring tools use the system built-in logging mechanisms. The connection between these is well defined and proven. It is therefore of help for the system administrators if these built-in logging mechanisms are used by all applications, as this allows the administrators to make use of existing tools (e.g. Nagios [Nagios 2014] or HP OpenView [Openview 2014] that collect, centralize, and search the logs [Limoncelli 2011].

The built-in system logging mechanisms take care of the log file location problem. They also prescribe the format, thereby forcing the developers, but also supporting them, to make consistent use of logging on the appropriate granularity.

It is also a lot easier to automatically generate incidents from specific defined events from the built-in system log for an IT service management (ITSM) tool. This ITSM tool can be configured to forward the automatically generated incidents directly, without human intervention, to the second line specialists. This way incidents are more easily solved without less human intervention, saving valuable time of the system administrators.

Of course logging in many cases has to be activated from within the system, so developers often have to explicitly program it into the system. But using the built-in logging mechanism alone does not ensure that the developers also make use of logging when it is appropriate. To address this issue guidelines could be defined and used by the developers for including logging in the system.

Now the alternative solution will be treated. If it is not possible to use the built-in system logging, e.g. because of different operating systems being used, then develop your own DIAGNOSTIC LOGGER [Harrison 2011] and define a standard for your system landscape that works well combined with the administration tools being used. Use the properties of built-in system logging mechanisms as basis for the requirements of your own logging mechanism. The most important point hereby is that this mechanism can be connected to the ITSM tools used by the system administrators. Ensure that this standard system is used for logging. This approach can be combined with SINGLE FILE LOCATION. Another solution could be making use of "logging as a service" providers, one just forward the syslog or agents and the log service providers will do the collecting of the data. An example of a log service provider is papertrail [Papertrail 2014].

Some requirements a good log should met to be valuable are ([anti patterns 2014]):

—Log actions before they happen.
—Mind the file size if logs should be copied or archived.
—Split messages into different files depending on intended audience/way of using.

On the type of usage it depends how robust the chosen solution acts within daily use:

—When normal availability is desirable one can choose, when the centralized logging system fails, to recover the server and reload the logging of the several subsystems.
—When high availability is needed the centralized logging system needs to be made so, e.g. as a High Availability cluster.

Otherwise the chosen solution could become a Single Point of Failure (SPoF).

For implementing a (system) logging facility one can make use of FACTORY [Gamma et al. 1994], which makes it possible to create loggers (ADAPTERS [Gamma et al. 1994]) for different applications. See figure 5.

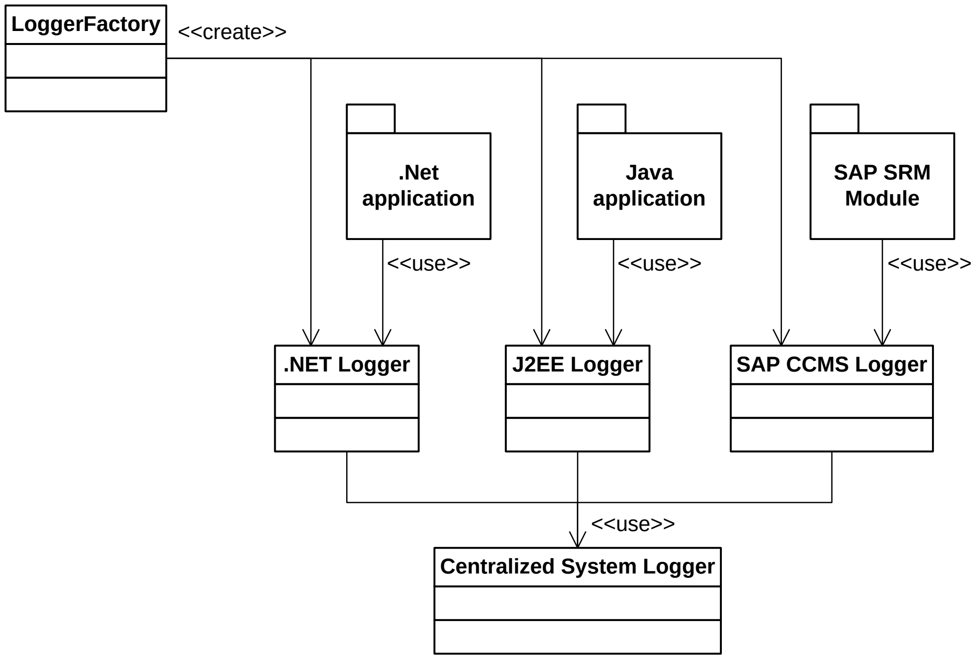

Fig. 5. Alternative solution structure of CENTRALIZED SYSTEM LOGGING

❖ ❖ ❖

Rationale:
Becasue one wants just one instance of a system logger, SINGLETON [Gamma et al. 1994] seems to be the preferred way to implement it. But becasue one wants to be able to test it, beside the production version, several instances should be possible therefore the number of instances should be parametrizable.

Known Uses:
Many monitoring tools provide a mechanism for gathering several logs to one central place, but even easier to use is a distributed log collector:

—Scribe ([Scribe 2014]) is a scalable log aggregation server used and released by Facebook as open source. Scribe is written in C++ and uses Thrift ([Thrift 2014]) for the protocol encoding. Since it uses thrift, virtually any language can work with it.
—Flume ([Flume 2014]) is an Apache project for collecting, aggregating, and moving large amounts of log data. It stores all this data on HDFS ([HDFS 2014]).

As an example of the implementation of CENTRALIZED SYSTEM LOGGING we have performed with our second year students System and Network Engineering some practical scripting exercises with Python where they, amongst others, use some standard libraries available for Python to log events to the system event log and afterwards create a statistical plot of it with the help of the Python library Matplotlib. An example of a call from Python to the Windows system log is:

```
import win32evtlogutil
win32evtlogutil.ReportEvent(ApplicationName, EventID, EventCategory,
                            EventType, Inserts, Data, SID)
```

This way the students get a feeling for how to integrate information from several resources (systems and applications) into one central store (system event log) and transform that information into a graphical output which could give insighs into e.g. the number of incidents per month with error level ERROR.

3. CONCLUSION

In the article *A plea from sysadmins to software vendors: 10 Do's and Don'ts* by Thomas Limoncelli [Limoncelli 2011], system administrators collected a basic list of do's and dont's for software vendors in order to make the life of the system administrators more easy.

This paper aims at influencing the software architects and the software architecture by providing patterns for software architecture that are endorsed by system administrators.

Both approaches — software architecture patterns for realizing the above described quality attributes and patterns that support the work of system administrators — don't touch some important aspects of the intersection of software architecture and system administration. Therefore we want to introduce a set of patterns which bridges this gap, based on the needs of the system administrators.

The problems that are cited in the aforementioned article have been experienced within daily system administration practice.

Further patterns we want to work on are CENTRALIZED IDENTITY MANAGEMENT and MULTI-TENANCY.

With this starting point for a repository of this kind of patterns we want to give ideas to software architects and application developers on how to improve their applications from a system administration viewpoint. Beside these patterns we want to bridge the gap between system administrators and the software architects of the software which needs to be administered by these system administrators.
We think it is interesting to look for a specialized modelling language like SysML [SysML 2014] to describe these patterns in the future.

4. ACKNOWLEDGEMENTS

We wish to express many thanks to our shepherd Eduardo Guerra for all his useful comments and feedback. Furthermore the feedback given by the participants of the writers' workshop at the PLoP'13 conference helped a lot to improve this work.

REFERENCES

ALEXANDER, C., ISHIKAWA, S., AND SILVERSTEIN, M. 1977. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press.

ANTI PATTERNS, L. 2014. logging-anti-patterns.

BUSCHMANN, F., HENNEY, K., AND SCHMIDT, D. C. 2007. *Pattern-oriented software architecture: On patterns and pattern languages*. Vol. 5. John Wiley & Sons Inc, Chichester, UK.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. 1996. *Pattern-oriented Software Architecture - A System of Patterns*. John Wiley & Sons, Chichester, Chichester, UK.

FHS. 2014. Filesystem Hierarchy Standard. `http://www.pathname.com/fhs/`. Accessed: 25-04-14.

FLUME. 2014. Flume. `http://flume.apache.org/`. Accessed: 25-4-2014.

FOWLER, M. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1994. *Design Patterns: elements of reusable object-oriented software.* Addison-Wesley, Boston, MA.

GOOGLE APPS PLATFORM. 2014. Google Apps Administrative APIs -. Accessed: 20-04-14.

HANMER, R. 2007. *Patterns for Fault Tolerant Software*. Wiley Publishing.

HARRISON, N. B. 2011. Improving quality attributes of software systems through software architecture patterns. Ph.D. thesis, Rijksuniversiteit Groningen, Groningen.

HDFS. 2014. HDFS. `http://hadoop.apache.org/`. Accessed: 25-4-2014.

HOHPE, G. AND WOOLF, B. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

HUMBLE, J. AND FARLEY, D. 2010. *Continuous Delivery*.

KIRKPATRICK, K. 2013. Software-defined networking. *Communications of the ACM 56,* 9, 16–19.

LIMONCELLI, T. A. 2011. A plea from sysadmins to software vendors: 10 Do's and Don'ts. *Communications of the ACM 54,* 2, 50–51.

MICROSOFT. 2013. Choose the right API set in SharePoint 2013. http://msdn.microsoft.com/library/office/jj164060.aspx. Accessed: 20-04-14.

NAGIOS. 2014. Nagios. http://www.nagios.org/. Accessed: 25-April-2014.

OIAM. 2014. OIAm Repository. https://www.infra-repository.org/oiar/index.php/Main_Page. Accessed: 20-04-2014.

OPENVIEW. 2014. HP Openview. http://en.wikipedia.org/wiki/HP_OpenView. Accessed: 25-4-2014.

ORACLE. 2014. Java Management Extensions (JMX). http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html. Accessed: 20-04-14.

PAPERTRAIL. 2014. Papertrail. https://papertrailapp.com/. Accessed: 25-4-2014.

PASCHKE, A. AND SCHNAPPINGER-GERULL, E. 2006. A Categorization Scheme for SLA Metrics. In *Service Oriented Electronic Commerce*, M. Schoop, C. Huemer, M. Rebstock, and M. Bichler, Eds. LNI Series, vol. 80. GI, 25–40.

RAILS. 2014. Ruby on Rails. http://rubyonrails.org/. Accessed: 25-04-14.

SCHUMACHER, M., FERNANDEZ, E., HYBERTSON, D., BUSCHMANN, F., AND SOMMERLAD, P. 2005. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons.

SCRIBE. 2014. Scribe. https://github.com/facebook/scribe. Accessed: 25-4-2014.

SYSML. 2014. OMG SysML. http://www.omgsysml.org/. Accessed: 25-04-14.

THRIFT. 2014. Thrift. http://thrift.apache.org/. Accessed: 25-4-2014.

WELLHAUSEN, T., WAGNER, T., AND MÜLLER, G. 2010. Handling Application Properties.