Software Architecture Patterns for System Administration Support

ROLAND BIJVANK, HU University of Applied Sciences, Utrecht, the Netherlands
roland.bijvank@hu.nl

WIEBE WIERSEMA, HU University of Applied Sciences, Utrecht, the Netherlands
wiebe.wiersema@hu.nl

CHRISTIAN KÖPPE, HAN University of Applied Sciences, Arnhem, the Netherlands
christian.koppe@han.nl

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 as important as other aspects. In this work we start with mining software architecture patterns that, when applied by software architects, support the work of system administrators. We present five patterns: PROVIDE AN ADMINISTRATION API, SINGLE FILE LOCATION, USE BUILT-IN SYSTEM LOGGING, CENTRALIZED IDENTITY MANAGEMENT, and MULTI-TENANT APPLICATION.

Categories and Subject Descriptors: D.2.10 [**Software Engineering**]: Design—*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), 11 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 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.

For a large number of points in this list there is a high agreement between administrators on what the best practices should be. However as system adminstrators are on the "receiving" end for a new or modified application, it is necessary to influence other parties who have a key position in the creation or the changing of 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. The design decisions taken in that document have a profound impact on the workload 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.

The focus of Software Architecture is often on realizing quality attributes, such as those described in ISO 25010: Functional suitability, Performance efficiency, Compatibility, Usability, Reliability, Security, Maintainability and Portability. Many patterns have been described, e.g. in the POSA book series [Buschmann et al. 1996], and their general applicability

for realizing the qualities has been discussed [Harrison 2011]. There are also publications that focus on patterns for specific quality aspects, like patterns for fault tolerant systems [Hanmer 2007] or security patterns [Schumacher et al. 2005]. But there is one quality attribute where not much attention has been paid to: Portability and its sub-qualities Adaptability, Installability, and Replaceability. A number of concerns from system administrators are covered by the aforementioned attributes, but the mapping of the concerns on the attributes is not intuitive.

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)[1] - 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. The Building Blocks Model is primarily a *decomposition* tool. That means that it is used to dissect infrastructure landscapes into logical dimensions and parts in order to enable structured and methodological modeling (*composition*).

—Gregor Hohpe and Bobby Woolf: Enterprise Integration Patterns[2] - this site provides a consistent vocabulary and visual notation to describe large-scale integration solutions across many implementation technologies. It also explores in detail the advantages and limitations of asynchronous messaging architectures.

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.

With these patterns 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

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

—ADMINISTRATION API

—SINGLE FILE LOCATION

—BUILT-IN SYSTEM LOGGING

The patterns use a 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 and offer a rationale for the pattern based on literature.

[1] http://www.infra-repository.org/oiar/index.php/Main_Page

[2] http://www.eaipatterns.com/

ADMINISTRATION API

Every software system in a professional environment needs to be maintained by a system administrator. Most applications provide an administrative interface for system administrators to perform these tasks.

❖ ❖ ❖

If the administrative interface is a GUI, many of the standard administration tasks can not be 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 usage. System administrators have their own ways of organizing their administration tasks. They strive to automate many parts, often in unexpected ways, and a GUI is minimizing the possibilities of doing so.

Platform diversity. The operating systems which administrators are using for their administration tasks often 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. The increase of usage of a resource when technology improves the resources efficiency was shown in 19^{th} century and called "*Jevons Paradox*" [Polimeni et al. 2008]. It seems quite likely the same will apply to the number of systems deployed in the Cloud, which in consequence will lead to a even higher workload for the system administrators.

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 and well documented, so that admins can automate administrative tasks and integrate it easily in the administration processes.

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 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, then also the API is 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. For example: an API might require the correct spelling of user roles which need to be assigned to new users. A GUI can offer a selection list including all user roles and possibly an extra explanation of these roles in an apart 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 subsystems that offer the administration functionality. This is shown in Figure 1.

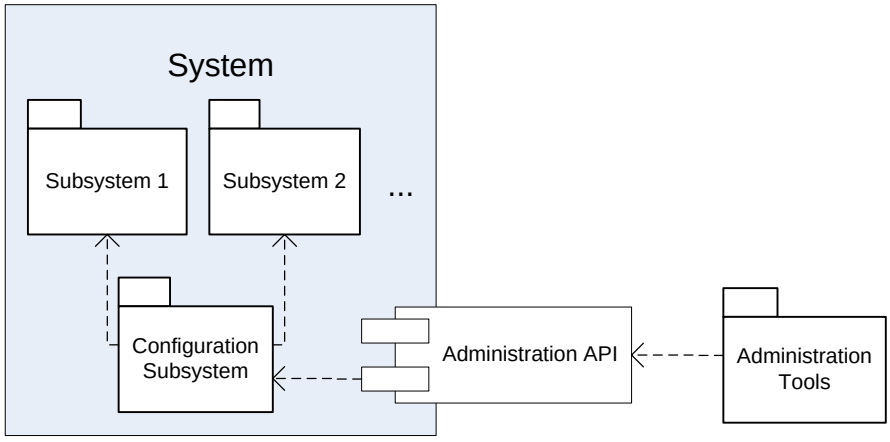

Fig. 1. Main solution structure of ADMINISTRATION API

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. These can be implemented making use of e.g. a pattern like ADAPTER [Gamma et al. 1994] because this pattern can influence the visibility of the administration API with respect to authentication and authorization of the requester.

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.

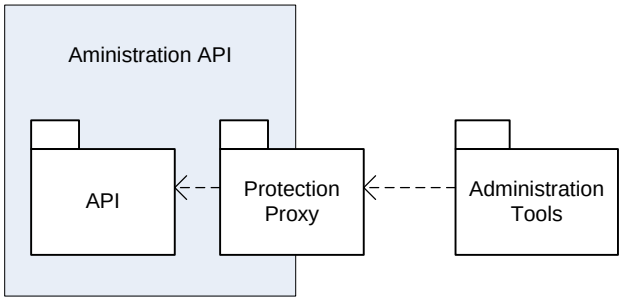

Fig. 2. An administration API including a protection proxy for security reasons

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.

This pattern is related to MAINTENANCE INTERFACE [Hanmer 2007], which states that one should separate application and maintenance requests into two different interfaces so if an application is overloaded it still can be maintained. The problem addressed by MAINTENANCE INTERFACE is to assure that the application still can be maintained even when it's overloaded, but it does not state *how* to do that best. ADMINISTRATION API addresses this *how* and therefore can be combined with MAINTENANCE INTERFACE.

One possibility of implementing this administrative API in the Java programming language are Java Management Extensions[3] *(JMX).*

[3] `http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`

SINGLE FILE LOCATION

Files are an old and established mechanism used by applications to store and retrieve configuration, libraries, state, data etc. Most applications 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.

❖ ❖ ❖

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.

—Hard-coded Locations.
This is the case when the 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.

—Non Human Readable Configuration Files.
When an application provides a Graphical User Interface for configuring the application, it happens that such an application stores the captured configuration in a non human readable file. The disadvantage of this approach is that the system administrator can only see the configuration by starting the application and opening the dialogue to see the settings. This also blocks automation of deploy and install scripts and integration with automatic deploy tooling such as Puppet[4] or Chef[5].

❖ ❖ ❖

Therefore: Put all related files in one (hierarchical) location. Make the path of this location configurable.

Analyze the files and folders of the application, group the files that logically belong together and should be at the same location 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 BUILT-IN SYSTEM LOGGING.

Ideally it should be a structure that is re-used across applications that are installed on the same the server.This provides consistency for the system administrators.

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

The applications that do file access should not use the native File IO Libraries but should use a FACADE for accessing files. 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

[4] `https://puppetlabs.com/`
[5] `http://www.opscode.com/chef/`

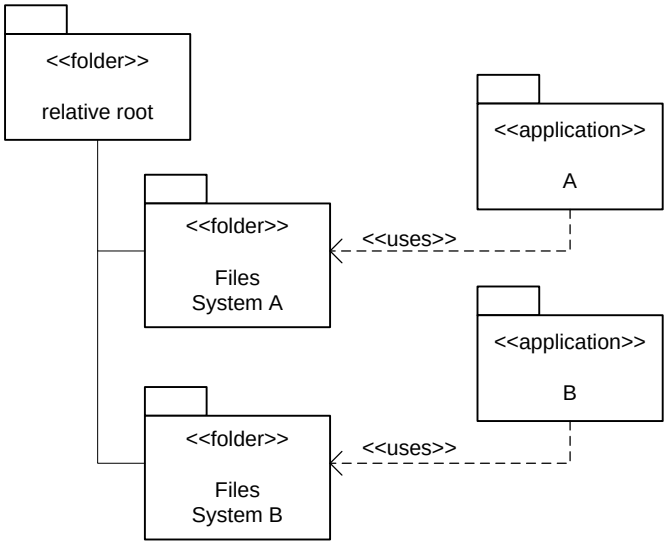

Fig. 3. The hierarchical file structure

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 facade.

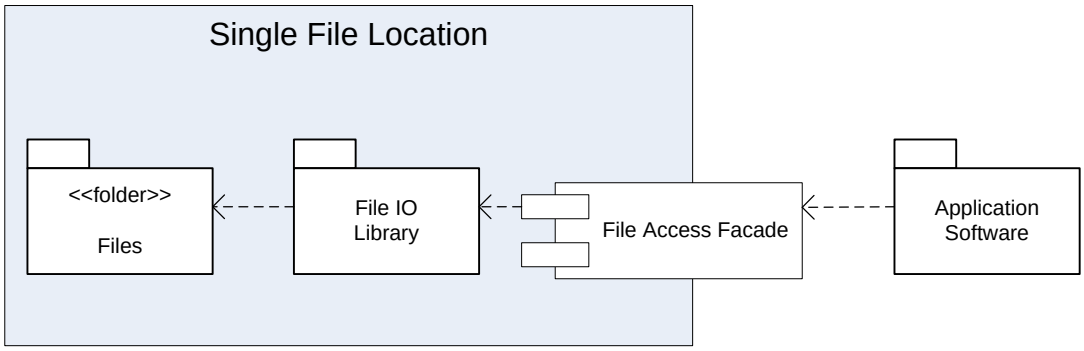

Fig. 4. Main solution structure for file access using SINGLE FILE LOCATION

❖ ❖ ❖

Rationale.

Without using this pattern the files of applications will be dispersed over several distinct locations which makes it hard to maintain the application. 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.

If the folder structures are standardized across a development group, developers find it easier to navigate across an application and find the right files and folders.

Using SINGLE FILE LOCATION is also more secure as this blocks access to vulnerable parts of the operating system as the implemented facade blocks access outside the root location. It is somewhat similar to a jailshell that prohibits users from wandering outside their home directories.

A nice example of the structure of SINGLE FILE LOCATION is for instance found in the way ruby of rails applications are structured. Every project starts with a pre-defined folder and file structure.

BUILT-IN SYSTEM LOGGING

The application needs to provide the ability of logging certain events or actions by using the built in system logging of a platform.

❖ ❖ ❖

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.

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 difficult to gather those files to one stack.

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.

❖ ❖ ❖

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

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 or HP OpenView) 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 woithout less human in tervention 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.

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 2001] and define a standard for your system landscape that works good in combination 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.

Some requirements a good log should met to be valuable are:

—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.

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.

For a large number of points in this list there is a high agreement between administrators on what the best practices should be. However as system adminstrators are on the "receiving" end for a new or modified application, it is necessary to influence other parties who have a key position in the creation or the changing of 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. The design decisions taken in that document have a profound impact on the workload 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.

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:

—Open Infrastructure Architecture repository (OIAr)[6]
—Enterprise Integration Patterns[7]

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.

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.

4. ACKNOWLEDGEMENTS

todo

REFERENCES

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

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

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.

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

HARRISON, N. B. 2001. Patterns for logging diagnostic messages. 173–185.

HARRISON, N. B. 2011. Phd. Ph.D. thesis, Rijksuniversiteit Groningen.

[6] http://www.infra-repository.org/oiar/index.php/Main_Page
[7] http://www.eaipatterns.com/

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

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.

POLIMENI, J. M., MAYUMI, K., GIAMPIETRO, M., AND ALCOTT, B. 2008. *The Jevons paradox and the myth of resource efficiency improvements*. Vol. 18. Earthscan.

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

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