Emerging Patterns of Continuous Integration for Cross-Platform Software Development

Chin-Yun Hsieh
Department of Computer Science and Information Engineering
National Taipei University of Technology
Taipei, Taiwan 106
hsieh@csie.ntut.edu.tw

Yu Chin Cheng
Department of Computer Science and Information Engineering
National Taipei University of Technology
Taipei, Taiwan 106
yccheng@csie.ntut.edu.tw

Chien-Tsun Chen
Software Department
Super Micro Computer, Inc. Taiwan
Taipei, Taiwan 235
ctchen@ctchen.idv.tw

ABSTRACT

This paper proposes a collection of continuous integration patterns for use in developing cross-platform software. The patterns reflect our experience in building commercial and open-source cross-platform software which made extensive use of continuous integration systems. We focus on the patterns that represent the basic types of projects to be put on a continuous integration system to produce cross-platform software. An example is given in the pattern language form to illustrate the use of these patterns. By putting these patterns in a pattern language, insights on the relationships among these patterns become apparent. By applying the patterns in this pattern language, continuous integration can be made to support the development of cross-platform software better.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]

General Terms

Design

Keywords

Pattern, Pattern Language, Continuous Integration, Cross-Platform

1. INTRODUCTION

Continuous integration plays a crucial role in the success of iterative and incremental software development. It serves as an information radiator for the team to monitor the current state of the product under development. Consistently practiced, continuous integration can help guaranteeing that working software is delivered at the end of an iteration/increment. As such, continuous integration needs to be made a sustainable practice in incremental and iterative software development. This paper takes the pattern approach to address at the issues of planning and executing continuous integration from the perspective of *cross-platform software development*. Here, cross-platform software means software products that simultaneously support multiple operating systems, e.g., Microsoft Windows and various Linux distributions. Further, we limit the scope to cross-platform software that contains both *platform-dependent* and *platform-independent* components.

While cross-platform software harbors the potential for software developing organizations to expand markets and earn more profit, it is also more costly to develop. In our experience, most of the extra cost has come from the increased integration and testing efforts. Therefore, the ability to harness automatic integration and testing technologies is essential for cutting the cost of developing quality cross-platform software. To this end, within continuous integration, a widely used software practice in the agile community, lies the promise for harnessing automatic integration and testing technologies. However, to development teams that have little or no prior experience in practicing continuous integration, cross-platform issues present even greater challenges. To effectively take up the challenges, the development team will need a common language of cross-platform continuous integration to analyze, design, discuss, implement, and improve its continuous integration practices. This paper presents the first results of our attempt to establish such a common language.

The patterns proposed here reflect our working experience in both academic and industrial software development. Some of the patterns presented are quite basic and can be applied to software development in general, while others are more specific in dealing with cross-platform issues. By putting them together in a pattern language, a complete picture of continuous integration for cross-platform software development is depicted. By applying the patterns, continuous integration practices can be made sustainable and the cost for developing of cross-platform software can be reduced.

So far, we have identified eleven patterns. Following the convention set in the Gang of Four book [3], these patterns are organized into a catalog consisting of patterns belonging to three categories: *Project*, *Build*, and *Good Habit*; see Table 1. Patterns in the *Project* category deal with subdivision of a large project of cross-platform continuous integration into sub-projects based on the principles of *separation of concern* and *module decomposition* [14]. Patterns in the *Build* category deal with workflow of continuous integration and strategies for preventing broken builds. Lastly, patterns in the *Good Habit* category suggest responsibility assignment among the developers in dealing with a broken build. In this paper, we shall focus on the six patterns in the Project category; Section 2 gives a usage example. The first two patterns, *Installation Project* and *Single Shared Library Project*, are presented in a fully-dressed format in Section 3; they constitute the target for shepherding in this workshop. The remaining patterns are included as supporting materials and are briefly presented as problem-solution pairs in Section 4. Related work is given in Section 5. The paper concludes in Section 6.

Table 1. Pattern categories

Category	Patterns
Project	Installation Project, Single Shared Library Project, Interface Project, Cross-Platform Project, Native Project, Patch Project
Build	Local Private Build, Remote Private Build, Cross-Platform Integration, Integration Workflow
Good Habit	Single Responsible Person

2. PATTERNS IN ACTION

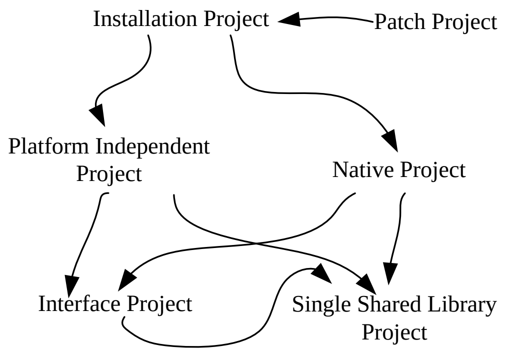

Figure 1. Relationships of patterns for planning projects of cross-platform continuous integration.

You are in charge of developing a software product called ezMonitor for monitoring the health of a computer system. While developed mainly with the Java language, ezMonitor also includes C/C++ code for accessing the computer system's health information collected by the hardware sensors on the motherboard controlled by a number of health chips. For example, ezMonitor can monitor CPU temperature and fan speeds.

To get health readings, ezMonitor communicates with the health chips through low-level chip drivers. In the past, vendors of these heath chips regularly provided drivers for the Windows platform but rarely did so for the Linux platform. As a result, motherboard vendors, being under-supported on the Linux platform, usually provided health monitoring systems only for the Windows platform. The circumstance has changed recently. Nowadays, Linux-based drivers are very common (either built-in in the Linux kernel or provided by vendors). To take advantage of the new circumstance, your company has decided to develop a cross-platform version of ezMonitor.

Before writing any code, you need to design the overall structure of the product. By applying the general principles of *separation of concerns* and *module decomposition* [14], you decide to divide the whole product into a number of (sub-) projects. Fig. 1 shows the decomposition and the relationships among the projects. Each type of the projects plays a different role.

First, you create an *Installation Project* that produces an easy-to-use installer that can help users to smoothly install ezMonitor on their systems. Next, you create *Platform Independent Projects* and *Native Projects*. The former include hardware and operating system independent programs such as user interfaces and application control modules; the latter contain hardware and operating system related programs such as drivers for health chips and dynamic-link libraries. Since your product needs to support both Windows and Linux-based systems, the *Platform Independent Projects* should not directly invoke any services provided by operating systems. Thus, you create *Interface Projects* that define the interfaces between the *Platform Independent Projects* and the *Native Projects*. The *Platform Independent Projects* use Platform Factory [1] defined in the *Interface Projects* to access a platform-specific implementation of an interface. Platform-specific implementations are defined in the *Native Projects*.

After the *Platform Independent Projects*, *Native Projects*, and *Interface Projects* are successfully built, one or more files are generated. Each of a *Platform Independent Project* and an *Interface Project* produces two jar files: one containing Java bytecode and the other containing Java source code. A *Native Project* produces dynamic-link libraries (.dll or .so files), drivers, and platform-dependent shell scripts.

With ezMonitor divided up into a number of projects, the built outcomes of a project are referenced or reused by the other projects. In particular, the *Installation Project* needs the built outcomes from these divided projects to produce installation programs. You create a *Single Shared Library Project* which acts as a repository that contains the build outcomes. Note that the *Single Shared Library Project* does not contain any build script and does not need to be added into a continuous integration system. The *Platform Independent Projects*, *Interface Projects*, and *Native Projects* reference the *Single Shared Library Project* and actively publish their build outcomes to it. The *Single Shared Library Project* also contains third-party libraries such as the Apache log4j (http://logging.apache.org/log4j/1.2/) and the Apache commons CLI (http://commons.apache.org/cli/). By requiring projects that use shared libraries to always use a copy from the *Single Shared Library Project*, it is ensured that different projects that share a same library will always use the same version of that library.

Lastly, look at the *Installation Project* again. Once the implementations of the projects (i.e., *Platform Independent Projects*, *Native Projects*, and *Interface Projects*) are ready and can be built and integrated successfully, an installer can be produced by packaging necessary files from these projects. You can also add a *Patch Project* to produce quick fixes from the build outcomes of two consecutive versions produced by the *Installation Project*. By applying the quick fixes, users can update to the new version without reinstalling the whole software.

3. TWO PROJECT PATTERNS

3.1 Installation Project

Context: Traditionally, deployment concerns of a software product are considered at the late stage of the software development, usually before release. For a project that applies incremental and iterative processes as well as continuous integration, working software is expected to be ready at the end of iteration. The working software may be deployed internally for testing and for demonstration. It may also be deployed on the customer site for trial use. Since multiple target platforms are involved in a cross-platform software product, the deployment task is made much more difficult.

Problem: How to provide an easy-to-use and robust method for users to deploy a cross-platform software product?

Forces:

- A software product can fail to attract users if it cannot be easily installed.
- Different operating systems have their own flavors in installation modes, ranging from text-based to GUI-based modes and using different formats such as the Microsoft MSI and the RedHat RPM.
- You want to simplify the installation procedures and minimize possible deployment problems.

Solution: At an early stage of software development, create an installation project to produce an installer for the cross-platform application under development. Add the project to your CI system. When the CI system builds the installation project either automatically or by request, an installer is produced. Release software at the end of iteration has become a norm for projects applying incremental and iterative development. Thus, it is reasonable to consider the installation problem at the early stage of product development so that working software can be provided via the installer. Once the installation project can produce an installer, find a way to automate the testing of the installer on all supported operating systems.

Many commercial and open source tools are available for creating installation programs. Some of them support the creation of cross-platform installers while the others are platform-specific. An example of the former category is InstallAnywhere [2]. By using InstallAnywhere, an *Installation Project* can produce multiple installers for multiple platforms. For example, InstallAnywhere can produce .exe installers for Windows platforms and .bin installers for Linux-based platforms. The installers can be configured in a way that they run in the GUI mode on Windows platforms and in the text mode on Linux-based platforms. InstallAnywhere also supports a silent installation mode. When an installer is run in the silent mode, users need not (and cannot) interact with the installer to modify the default behavior of the installer (e.g., to change the installation folder). Instead, the installer accepts a pre-defined configuration file that contains all necessary settings to guide the installer.

Developers can also use platform-specific tools to produce platform-specific installers. For example, InstallShield is a popular tool that prodcues .msi installers for the Windows platforms [2]. For Linux-based systems such as RedHat and Ubuntu, tools to package .rpm and .deb modules are used.

It is also possible to package a cross-platform product as the so-called green software, portable applications, or portable software if the product does not contain low-level native drivers which must be installed in a specific location and does not require the systems to reboot. Green software does not require a sophisticated installation procedure. To use the software, merely copy the software (usually an executable file or a folder) or unzip it to a hard disk or flash disk.

Resulting Context: Sophisticated tools like InstallAnywhere are convenient for producing cross-platform installers. However, such a tool is usually very expensive. Also, the produced installers may be larger in size than those produced by platform-specific installation tools. Sophisticated installation tools that support complicated scripts and customization functions produce flexible installers, but can lead developers to experience a sharp learning curve and consequently put off the creation of an *Installation Project*.

Installation Projects produce installers to deploy a software product from scratch. To install a new version of the software with the installers, the old one usually needs to be uninstalled first and then the new one can be installed. If only a minor part of the product is revised, using *Installation Projects* to produce an installer to upgrade the software may be taxing. In this situation, use a *Patch Project* instead.

Related Patterns: Installers package components produced by *Platform Independent Projects*, *Interface Projects*, *Native Projects*, and other components made available through a *Single Shared Library Project.* Apply *Cross-Platform Integration* to test an installer on all supported platforms.

3.2 Single Shared Library Project

Context: A software product intended either for a single platform or multiple platforms is usually structured into a number of interdependent projects. One common type of interdependency occurs when these projects share one or more common libraries. For example, ezMonitor includes both Java and C/C++ projects. In the projects written in Java, Apache Log4J is used. ezMonitor also includes two common libraries that are developed in-house: FileCommons for file manipulation and NetworkCommons for network operation. Some of the ezMonitor projects make use of both libraries; some others make use of one of them and still others make use of none. One way to share the libraries is to manually copy them to the project workspace. However, doing so complicates the version control of the shared libraries. For example, once a shared library is upgraded, each of the local copies needs to be upgraded manually as well. This can be error-prone and tedious for a product that has a number of projects sharing common libraries.

Problem: How do we make sharing common third-parity and in-house libraries among projects easy?

Forces:

- You want to use a single version of a shared library among the projects that use the library.
- Shared libraries, either third-parity or in-house, are likely to change over time.

- The version control mechanism for shared libraries should be as easy to use as possible.

Solution: Create a project that acts as a centralized repository for shared libraries. Organize the project structure in a way that third-parity libraries, in-house libraries, native (platform-dependent) libraries as well as system drivers, and source code of the libraries (if available) are placed in different folders. Define the following rules for each type of the libraries: (1) how should these libraries be referenced by other projects; (2) when and who can update a new version of a library; and (3) how does the library update process get initiated (e.g., manually or automatically).

Fig. 2 shows an example to organize a *Single Shared Library Project* named SharedLibraries. The SharedLibraries project acts as a centralized repository for a cross-platform product written in Java and C/C++. There are six different folders in the SharedLibraries project:

- driver: This folder contains system drivers, primarily for the Windows operating systems. Usually, a Windows system needs to be rebooted to enable the driver.
- native: This folder contains platform-dependent libraries (usually .dll files for Windows and .so files for Linux-based systems) or executable utility programs (e.g., the cfg_x64 program in Fig. 2).
- in-house: This folder contains the binary code of common in-house libraries that will be used by other projects. For example, if the MyServer project needs to use the FileCommons.jar file, it does so by directly referencing FileCommons.jar from the SharedLibraries. Whenever the FileCommons.jar is updated to a newer version, the MyServer project (and all projects that use the FileCommons.jar) can get the latest version of the FileCommons.jar by simply issuing an “update” command to sync its local data from the code repository.
- in-house-src: This folder contains the source code of common in-house libraries. The main purpose of placing library source code in the SharedLibraries project is for debugging and tracing. When everything goes well, developers usually do not want to know the internal implementation of a library. However, when things go wrong and an exception is raised, developers may need to locate the problem by following the stack trace. In this situation, jumping into the source code along the stack trace is necessary. If the stack trace contains an invocation to a method in the library, developers need its source code to explore the method implementation. An IDE (Integrated Development Environment) like Eclipse allows developers to link the source code of a library to its binary code so that the IDE can jump into the implementation code of the library while debugging.
- 3party: The function of this folder is similar to the in-house folder except that this folder contains libraries from third-parties.
- 3party-src: The function of this folder is similar to the in-house-src folder except that this folder contains source code of third-party libraries.

Resulting Context: One drawback of sharing libraries via a *Single Shared Library Project* is the “timing” issue when this pattern is applied in IDEs. Suppose that you use the Eclipse IDE to modify code in project A and project B at the same time and project A uses code in the project B. If project A references code in the project B via a *Single Shared Library Project*, modification in project B cannot be reflected immediately to project A. Modification in project B can be reflected in project A only after (1) a jar file (i.e., the library of project B) is generated and committed to the *Single Shared Library Project*; (2) project A syncs with the *Single Shared Library Project*. Thus, to a set of projects that are closely related, a *Single Shared Library Project* may not be suitable. In this situation, use project references in your IDE instead. A project reference example is shown in Fig. 3. Note that the *Single Shared Library Project* can still be used since project A is built after project B according to their dependency relationships.

Related Patterns: The build outcomes of *Platform Independent Projects*, *Interface Projects*, and *Native Projects* are stored in a *Single Shared Library Project.*

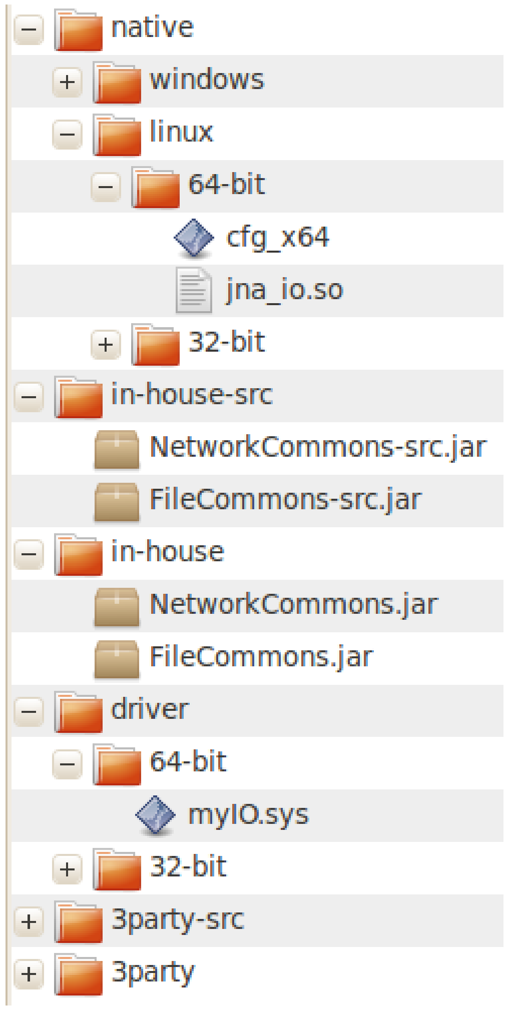

Figure 2. An example of a *Single Shared Library Project*

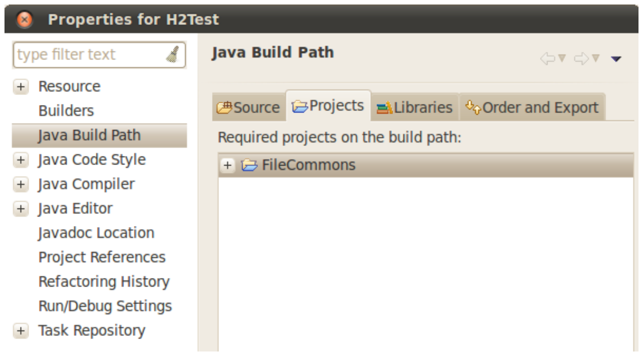

Figure 3. A screenshot of the Eclipse project reference.

4. THE SUPPORTING PATTERNS

4.1 Interface Project

Problem: How to isolate platform and implementation specific code from platform-independent projects?

Solution: **Define programming interfaces to encapsulate platform-dependent services or to allow a service to have multiple implementations. Put the interface definition code in an interface project. Do not mix interface definition code with interface implementation code.** "Programming to an interface, not an implementation" [3] and "dependence injection" [4] are well-know software development practices that require clearly defined interfaces. For cross-platform software, strictly separating interfaces from the implementation hides platform-dependent details and enhances the flexibility and maintainability of software systems [1][3].

4.2 Cross-Platform Project

Problem: How to organize platform-independent code?

Solution: Classify implementation code into two types: platform-dependent and platform-dependent. Put platform-independent code in cross-platform projects. From the continuous integration perspective, cross-platform projects are easier to build and test. They can be built and tested by using virtual machines such as VMWare, VirtualBox, and Xen. Developers can quickly verify cross-platform projects by making a *Local Private Build* before the projects are committed for integration.

4.3 Native Project

Problem: How to organize platform-specific code?

Solution: Classify implementation code into two types: platform-dependent and platform-dependent. Put platform-dependent code in native projects. Native projects can be built on virtual machines if they depend on operating systems rather than on hardware. For hardware dependent native projects, a *Remote Private Build* can be used before they are committed for integration.

4.4 Patch Project

Problem: How to produce quick fixes to resolve problems without reinstalling the whole software?

Solution: Create a patch project to generate patch files. One way to produce a patch file is to perform a "diff" on the two different software versions and package the difference in the patch file. The patch file also contains a script to upgrade necessary files on the system to be patched.

4.5 Local Private Build

Problem: How to prevent a broken build?

Solution: Build a project locally before it is committed. You can make a local build by pressing a build button in your IDE or by running a pre-defined build script. Using the same build script executed by the continuous integration system to make a local build can further reduce the possibility of causing a broken build.

4.6 Remote Private Build

Problem: How to prevent a broken build?

Solution: Use a remote agent to build a project before it is committed. The remote agent's platform provides an environment that is required to build the project. This pattern is usually used to verify a *Native Project* that requires a build environment that is different from the local development environment. Continuous integrations systems such as JCIS, CruiseControl, Hudson, Buildbot, TeamCity, and Bamboo support this pattern [5][6][7][8][9][10].

4.7 Cross-Platform Integration

Problem: How to build cross-platform products?

Solution: Use a continuous integration system that supports cross-platform integration. Apply *Integration Workflow* to guide the continuous integration system to dispatch projects on suitable remote platforms for integration. Continuous integration systems supporting this pattern include JCIS, CruiseControl, Hudson, Buildbot, TeamCity, and Bamboo [5][6][7][8][9][10].

4.8 Integration Workflow

Problem: How to manage the continuous integration process for a cross-platform product containing interdependent projects?

Solution: Design integration workflows to control the continuous integration process. Two types of essential integration workflows are concerned: intra-project and inter-project. The former decides which integration activities (e.g., compilation, testing, test coverage analyzing, and packaging) to be included in a build and the execution order of each integration activities. The latter decides the build order of all projects according to the relations of project references.

4.9 Single Responsible Person

Problem: How to deal with a broken build?

Solution: Assign a person in your team who will be notified once a build was broken. This pattern prevents broken builds from being ignored; especially those caused by overnight build scripts.

5. RELATED WORK

Although the general idea of continuous integration has been applied in software development for decades, the knowledge of continuous integration is not commonly captured in the pattern format. The famous Portland Pattern Repository [11] lists eleven continuous integration patterns: *SingleUnifiedBuildScript*, *UseOneCodeLine*, *CommitEarlyAndOften*, *ReduceSizeOfCheckIn*, *UpdateOftenCommitOnlyAfterTesting*, *IncrementalIntegration* *ContinuousIntegrationRelentlessTesting*, *SingleIntegrationPoint*, *CollectiveCodeOwnership*, *DontIntegrateMidTask*, and *SingleReleasePoint*. Most of the patterns are still under development. Useful materials to master continuous integration skills can be found in [12][13]. Continuous integration systems supporting cross-platform software development include [6][7][8][9][10].

6. CONCLUSION

In this paper, we proposed a number of continuous integration patterns for cross-platform software development. An example of applying the proposed patterns to organize cross-platform projects was demonstrated. Two patterns were presented in full format for shepherding. The proposed patterns have been captured, applied, and refined by the authors since the year of 2004. In our experience, these patterns have been very useful for planning and performing continuous integration in cross-platform software development. By sharing these patterns in their present form, we expect that they can be elaborated further and more people can use them to resolve continuous integration problems in developing cross-platform software. Also, we are currently collecting information about how continuous integration is done in popular cross-platform open source projects. Once the information is collected, a “Known uses” section will be added to each of these patterns to make them more complete.

7. ACKNOWLEDGMENTS

This work is funded by the National Science Council of Taiwan under the grant NSC99-2221-E-027-096-MY3. We thank our shepherd Dr. Takao Okubo for his constructive comments that lead to the improved presentation of this paper.

8. REFERENCES

[1] Logan, S., 2008. *Cross-Platform Development in C++: Building Mac OS X, Linux, and Windows Applications*. Addison Wesley, Upper Saddle River, N.J.

[2] FlexerSoftware, http://www.flexerasoftware.com/.

[3] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M., 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Reading, Mass.

[4] Fowler, M., 2004. Inversion of Control Containers and the Dependency Injection pattern. In http://martinfowler.com/articles/injection.html.

[5] Cheng, Y. C., Ou, P.-H. Chen, C.-T., and Hsu T.-S., 2008. A Distributed System for Continuous Integration with JINI, In *Proceedings of the 2008 International Conference on Distributed Multimedia Systems (DMS 2008)*, September 4-6, 2008, Boston, USA.

[6] CruiseControl, http://cruisecontrol.sourceforge.net/.

[7] Hudson CI, http://hudson-ci.org/.

[8] Buildbot, http://trac.buildbot.net/.

[9] TeamCity, http://www.jetbrains.com/teamcity/.

[10] Bamboo, http://www.atlassian.com/software/bamboo/.

[11] Portland Pattern Repository, http://www.c2.com/cgi/wiki?ContinuousIntegrationPatterns,accessed 2009.

[12] Duvall, P. M., Matyas, S., and Glover, A., 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*, Addison Wesley, Upper Saddle River, NJ.

[13] Humble, J. and Farley, D., 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison Wesley, Upper Saddle River, NJ.

[14] Bass, L., Clements, P., and Kazman, R., 2003. *Software Architecture in Practice*, 2nd Ed., Addison-Wesley, pp. 36-37.