Patterns for Continuous Integration Builds in Cross-Platform Agile Software Development

CHIN-YUN HSIEH¹ AND CHIEN-TSUN CHEN²
¹Department of Computer Science and Information Engineering
National Taipei University of Technology
Taipei, 106 Taiwan
E-mail: hsieh@csie.ntut.edu.tw
²Independent Software Consultant
P.O. Box 35-220 Taipei, Taiwan
E-mail: teddy.chen.tw@gmail.com

Cross-platform software development poses challenges to agile development teams in practicing continuous integration (CI) builds not only because such builds take a longer time to complete and are more likely to fail, but also because builds of different lengths and scopes must be available depending on the working circumstances. To deal with this situation, three aspects of build automation in CI – the structuring of source code modules, the management of intermediate and final build artifacts, and the execution of builds – must be re-considered to account for the cross-platform characteristics. This paper discovers and documents a collection of ten patterns of CI builds for use in developing cross-platform software in the three aspects re-considered. These patterns are distilled from known uses of builds in existing software and from our experience in building commercial and open-source cross-platform software. As illustrated with an example adapted from the development of a real-world commercial cross-platform software product, the patterns can be effectively applied to solve many commonly encountered problems in applying CI for agile cross-platform software development.

Keywords: continuous integration, software build, cross-platform software, pattern, pattern language, agile development

1. INTRODUCTION

Software development is an area notoriously known for frequent failures with scope and severity ranging from schedule slippage, project cancellation, requirement misunderstanding, low quality, and so on [1]. To combat the problems, agile development methods such as Scrum [2] and Extreme Programming [1] has been proposed. Agile methods advocate frequently release of good quality working software that is readily shippable to the customer. While many agile practices must be in place to create good quality working software on a regular basis, continuous integration (CI) [1] stands out as the most crucial practice. The act of CI is performed through software builds of various kinds. Typical software builds include compiling source code to obtain the executable program or components, compiling test code to obtain executable unit tests, running unit tests and acceptance tests to obtain test reports, performing static analysis to find problems in source code, and producing installable package from prebuilt components [1, 3-6]. Software builds are organized into ones with various scopes and lengths so that they do not impede the agile practices from being carried out by the agile team. For ex-
ample, during the team’s working hours, a build triggered by a code commit should be completed in a short time to avoid holding up the committing developer. Further, it should have a small and clearly defined scope to help the committing developer isolate the root-causes in case a build fails. On the other hand, during the hours when no one is actively writing and committing code, builds of a more comprehensive coverage can be performed since there is no problem of holding up the developers. Such a comprehensive build is necessary because it brings the whole team up to the same baseline when they begin their work the next day. CI practices are well-understood and have been documented in the form of patterns [7]. Patterns and pattern languages are an important way to document reusable solutions to recurring problems in object-oriented design and software architecture [8-10].

While how agile practices combat software development problem and how CI helps the agile practices are well-understood, the increasing adoption of cross-platform products, which is propelled by the explosive growth of mobile and cloud applications, introduces a new aspect that requires a close examination. Cross-platform software refers to applications built off a common code base that simultaneously supports multiple operating systems or platforms [11]. Cross-platform software brings additional constraints that can affect the development: not only a new release must cover all target platforms, but also each platform evolves independently at a different pace. Cross-platform software commonly includes both platform-independent code and platform-specific code [3, 11-14].

In this paper, enhancing on the existing guidelines and the component model [3, 4, 6, 15, 16], we propose a collection of patterns for planning, managing and executing CI builds in cross-platform agile development. In so doing, three aspects of build automation in continuous integration – the structuring of source code modules, the management of dependent, intermediate and final build artifacts, and the execution of builds – are accounted for the cross-platform characteristics. A real development case illustrates that applying the patterns allows an agile team to evolve the CI builds in developing a cross-platform product so that (1) the length of time required by a build is kept small enough so that feedbacks are obtained with little delay; that (2) scope of builds is factored so that a failed build affects as few developers as possible; and that (3) dependencies among the builds are maintained so that the developer who actually causes a failed build is notified to take charge.

The rest of the paper is organized as follow. Section 2 analyzes how cross-platform development aggravates software development problems and how proposed CI patterns help the agile practices in combating them. Patterns in the Artifacts Management category, the Source Code Modularity category, and the Build Execution category are presented in Section 3, Section 4 and Section 5, respectively. Section 6 discussed the results of applying the patterns in a real world development case – ezMonitor. Related work is given in Section 7. The paper concludes in Section 8.

2. THE PROBLEMS AND THE SOLUTIONS

We shall examine four typical problems encountered in software development including schedule slip, low quality and high defect rate, requirement misunderstanding and change, and few feedbacks and long delays between feedbacks. These problems are consolidated from the reference [1]. In each problem, we analyze how cross-platform
development aggravates the problems, explain how agile practices combat the problems, and argue how the proposed CI patterns help agile practices in combating them.

While the proposed CI patterns will be elaborated in Sections 3, 4 and 5, Table 1 concisely presents the patterns. Their relationships are depicted in the pattern map of Fig. 1. The collection of CI build patterns comprises ten identified patterns organized into three categories: Artifacts Management, Source Code Modularity, and Build Execution. Patterns in the Artifacts Management category deal with handling and sharing the artifacts created by building the modules as well as third-party libraries that are required by the builds. Patterns in the Source Code Modularity category deal with subdivision of source code into modules by applying the principles of separation of concern and module decomposition [9] to cross-platform software. Lastly, patterns in the Build Execution category deal with workflow of continuous integration and strategies for preventing broken builds.

Table 1. The proposed patterns.

<table>
<thead>
<tr>
<th>Category</th>
<th>Pattern</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artifacts Management</td>
<td>Installer</td>
<td>Create a platform-specific installation program for the cross-platform product.</td>
</tr>
<tr>
<td></td>
<td>Patch</td>
<td>Generate patches for an Installer.</td>
</tr>
<tr>
<td></td>
<td>Single Shared Library</td>
<td>Create a central repository for making shared libraries available to dependent modules.</td>
</tr>
<tr>
<td>Source Code Modularity</td>
<td>Interface Module</td>
<td>Define interfaces to encapsulate platform dependent services in an interface module.</td>
</tr>
<tr>
<td></td>
<td>Platform Independent Module</td>
<td>Put platform-independent code in Platform Independent Modules.</td>
</tr>
<tr>
<td></td>
<td>Native Module</td>
<td>Put native code in a Native Module.</td>
</tr>
<tr>
<td>Build Execution</td>
<td>Local Build</td>
<td>Perform a build locally. A Local Build is further distinguished as a Local Build – Developer or as a Local Build – CI.</td>
</tr>
<tr>
<td></td>
<td>Remote Build</td>
<td>Use a single CI system to deploy the build for a native module to remote platforms.</td>
</tr>
<tr>
<td></td>
<td>Integration Workflow</td>
<td>Design integration workflows to control build jobs.</td>
</tr>
<tr>
<td></td>
<td>Single Responsible Person</td>
<td>Assign one person in development team to get notified when a build is broken.</td>
</tr>
</tbody>
</table>

Fig. 1. The proposed CI Patterns for planning, managing and executing software builds for cross-platform agile software development.
Table 2 presents a concise list of agile practices whose definition can be found in references [1, 2].

| Agile practices to combat common software development problems. |
|---------------------------------|-----------------|-------------------|
| Short, time-boxed iterations    | definition of done | small tasks with frequent commits |
| working software                | iteration review  | 10-minute build    |
| feature (product) backlog       | iteration retrospective | daily build    |
| prioritized features            | product owner    |                   |
| small features                  | daily standup    |                   |

2.1 Schedule Slip

**Problem**: Deadline is close but no working software is produced. Cross-platform development aggravates the problem because working software must be tested, integrated, and made shippable on all supported platforms. Release is delayed if one among the platforms is behind schedule.

**Combating agile practices**: The development period is organized into multiple fixed-length *Short, time-boxed iterations* that run from 1 to 4 weeks that lead to releases of *working software* in short cycles, with preference given to high priority features decided by the *product owner*. Features worked on are *small* enough so that they can be completed within one iteration.

**How CI patterns help**: Ensuring that up-to-date working software is produced before the end of an iteration by including the creation of *installers* and *patches* on all supported platforms in the *integration workflow*.

2.2 Low Quality and High Defect Rates of the Product

**Problem**: Developers are likely to have different level of expertise and area of specialty. Left to the individual developer’s own discretion, quality becomes low and defect rates are high as a result. Cross-platform development aggravates the problem because each different platform adds an area of specialty and a defect can affect multiple platforms.

**Combating agile practices**: The team ensures quality of the *working software* through *definition of done*, which involves creating and maintaining comprehensive test suites for unit testing, integration testing and acceptance testing. Automated inspection of code and static analysis are performed as well. The test suites are re-executed and static analysis is performed after every change to the code base to guard against quality decay. Team revises their working *definition of done* in *iteration retrospective*.

**How CI patterns help**: Ensuring that tests and static analysis are performed timely and efficiently by staging *local builds* and *remote builds* of different scopes and lengths according to the *integration workflow*. 

2.3 Requirement Misunderstanding and Requirement Change

**Problem:** It is difficult to describe requirement in a way to ensure full understanding between the product owner and the development team. If the requirements take a long time to get implemented, the misunderstanding can be devastating. Also, requirement can change during the time the product is developed. Cross-platform development aggravates the problem because different platforms impose different sets of constraints; the platform evolution changes the constraints that must be considered along with the requirements.

**Combating agile practices:** An iteration begins with a planning meeting in which the developers work with the product owner to analyze the features to be completed and the constraints that must be satisfied for the current iteration. The completed features and the constraints they must satisfy are checked during the iteration review by the product owner to identify any discrepancies that exist.

**How CI patterns help:** The most recent installer is used to install the product for iteration review, enabling the product owner to decide if the newly added features satisfy the requirements and constraints. Concerns for different platform constraints are separated by keeping Native Modules and Platform Independent Modules separated through Interface Modules. Platform change is watched by the Single Responsible Person who has the exclusive authority to update the Single Shared Library.

2.4 Few Feedbacks and Long Delays between Feedback

**Problem:** At the development level, the developers do not know how well they do because there tend to be few feedbacks after they commit code. The same thing happens at the product level because there is usually a long delay in getting a feedback from the customer. Cross-platform development aggravates the problem because a build takes even a longer time to complete, discouraging developers to perform frequent builds, and hence the frequency of feedback is decreased.

**Combating agile practices:** The development team does daily standups so that everyone knows who’s doing what and what impediments there are; a feature is decomposed into a number of small tasks with frequent code commits; 10-minute build are performed before/at each commit; comprehensive daily build is performed so that the developers starts development with a clean baseline the next day; iteration review is performed at the end of iteration to get feedback from the product owner and customers.

**How CI patterns help:** The developer separated Platform Independent modules and Native Modules in Local Builds – Developer to get feedback with short delay; he/she uses Local build – CI guard against outdated components; he/she uses Remote Builds for building Native Modules. All these builds are organized in an Integration Workflow that creates Installers and Patches for all target platforms comprehensively.

### 3. ARTIFACTS MANAGEMENT PATTERNS

Artifacts Management patterns include Installer, Patch and Single Shared Library.
They are used to deal with handling and sharing the artifacts created by building the modules as well as third-party libraries required by the builds.

3.1 Installer

**Context:** An agile team delivers working software every iteration. The working software may be installed internally for testing and for demonstration. It may also be installed 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 complicated.

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

**Forces:**
- A software product can fail to attract users if it is not easily installable.
- Different operating systems use different installation procedures and different formats.
- It is desirable to simplify the installation procedures and to minimize possible installation problems.

**Solution:** In the first iteration of an agile process, create an Installer to produce an installation program for the cross-platform application under development. 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 [17] that can produce multiple installation programs from a single Installer project for multiple platforms, e.g., .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. Many tools support a silent mode that reads installation settings from a configuration file. They are ideal for automating installation.

Developers can also use platform-specific tools to produce platform-specific installers. For example, InstallShield is a popular tool that produces .msi installers for Windows platforms [17]. 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 after installation. 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:** Installer ensures that working software is produced on a regular basis and at the end of each iteration in agile development. Installer enables automatic functional testing to be performed on all supported operating systems and platforms.

Sophisticated tools like InstallAnywhere are convenient for producing cross-plat-
form 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. Though sophisticated installation tools that support complicated scripts and customization functions produce flexible installers, developers can experience a sharp learning curve and consequently put off creating an Installer project at the early stage of software development.

Installer projects produce an installation program for deploying a software product from scratch. To install a new version of the software with the installers, the old one usually needs to be completely uninstalled first and then the new one can be installed thereafter. If only a minor part of the product is revised, using Installer projects to produce an installer to upgrade the software may be taxing. In this situation, use a Patch project instead.

**Known Use:** Berczuk suggests that software should be installed early and often [18]. Logan recommends providing support for native installers for cross-platform projects [2]. eXtreme Programming (XP) encourages developers to deploy every tested change to production [1], which is much easier to be implemented with installers. Continuously releasing user-installable software is also a recommended practice [4].

**Related Patterns:** Installer packages artifacts produced by building Platform Independent Modules, Interface Modules, Native Modules, and other components made available through a Single Shared Library. Apply Remote Build to test an installer on all supported platforms. Creating an Installer requires building all modules and linking with external components, which requires a comprehensive Integration Workflow.

### 3.2 Single Shared Library

**Context:** A cross-platform software product is usually complex enough to warrant partitioning into a number of interdependent modules. The modules can be interdependent of each other and can share one or more common external libraries. For example, all of the Java projects in ezMonitor make use of Apache Log4J for logging exceptions. ezMonitor also includes two common libraries that are developed in-house: FileCommons for file manipulation and NetworkCommons for networking operations. Some of the ezMonitor projects make use of both libraries; some others make use of just one and still others use neither. One straightforward way to make the shared libraries available to the dependent projects is to manually copy them to the workspace of each dependent project. 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 to make sharing common third-party and in-house libraries among projects easy?

**Forces:**
- The same version of a shared library must be used among different modules.
• Shared libraries, either third-party or in-house, are likely to change over time.
• The version control mechanism for shared libraries should be easy to use as much as possible.

Solution: Create a central repository for making shared libraries available to dependent modules. Structure it in such a way that third-party libraries, in-house libraries, native libraries, system drivers, and archived source code (if available) of these libraries and drivers are placed in well-defined folders. Determine on the following rules for each type of the libraries: (1) how these libraries are referenced by other modules; (2) when and who can update a new version of a library; and (3) how the library update process gets initiated (e.g., manually or automatically).

Fig. 2 shows an example to organize a Single Shared Library for a cross-platform product written with Java and C/C++. There are six different folders in the project:

• The folder driver contains system drivers primarily for Windows operating systems.
• The folder native 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).

Fig. 2. An example of a Single Shared Library.

• The folder in-house contains the binary code of common in-house libraries used by the other modules. For example, a dependent project uses the FileCommons.jar file by directly referencing it. Whenever the FileCommons.jar is updated to a newer version, the dependent project obtains the latest version by simply updating against the central repository used by the Single Shared Library.
The folder *in-house-src* contains the source code of common in-house libraries. Source code is placed in the Single Shared Library project for debugging and code-tracing purposes. When all goes well, the developers will not want to know the implementation of a shared library. However, when things go wrong (*e.g.*, 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 becomes 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.

- The folder *3party* serves a similar purpose to the in-house folder except that it contains libraries from third-party vendors.
- The folder *3party-src* serves a similar purpose to the in-house in-house-src folder except that it contains source code of third-party libraries.

**Resulting Context:** The central repository of *Single Shared Library* is readily implemented with version control systems such as CVS and SVN. A dependent module updates against the central repository to obtain the latest shared libraries before it is built.

One drawback of sharing libraries via a *Single Shared Library* project is the “timing” issue when this pattern is applied through IDEs. Here is an example with the Eclipse IDE. When the code in project A and project B is modified at the same time and project A references code in project B via a *Single Shared Library* 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 updates against the *Single Shared Library* project. As can be seen, this creates a possible delay. Thus, if a set of projects are closely related, use project references in your IDE instead. A project reference example is shown in Fig. 3.

![Fig. 3. A screenshot of the Eclipse project reference.](image)

**Known Use:** Maven has a central repository hosted on the remote site repo.maven.apache.org that stores and manages a number of shared libraries [19]. Build jobs use Maven to automatically download not only a library, but also all of its dependencies from the cen-
Developers can create a local repository to cache downloads from a remote repository [15, 19]. To organize a cross-platform project in a source code repository such as CVS and SVN, Logan suggests using a lib folder to store prebuilt libraries [11]. When the project is built, necessary libraries are copied to the build job working folder from the lib folder. The component cache described by P. Smith is similar to the Single Shared Library [3].

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

3.3 Patch

Context: A number of bugs have been fixed. The bug fixes are to be made effective to a running system at a customer’s site. Although an Installer has been created, to use it the customer needs to remove the old program and then install the new one with the following extra work: the configuration and user data need to be backed up, the system needs to be shut down, and backup data must be restored after rebooting. Using a new installer merely to upgrade a small piece of function is both risky and taxing.

Problem: How to produce quick fixes to resolve problems without significantly impacting an existing system?

Forces:
- Minor changes (e.g., bug fixes, additional library supports, etc.) must be made available to an application that is in use.
- Some application data can be lost once the application is uninstalled.
- It is desired to minimize the system downtime when upgrading a software system.
- If a large number of client installations are in existence, remotely deploying a complete installer to the client sites can consume a significant network bandwidth.
- The software product is released in very short cycles.

Solution: Create a Patch to generate patches for an Installer. One way to produce a patch is to conduct a “diff” on two different versions of an application and package the difference in the patch program. The patch is either an executable program or an achieved file containing a script to upgrade necessary data such as binary executable, database schema, and configuration data of an application to be patched.

Resulting Context: Nowadays, open source applications and mobile phone apps are released frequently both due to time-to-market consideration and in responding to customers’ feedback. In such a context, a Patch is more suitable than an Installer. Other benefits and liabilities of the Patch pattern include the following:

- **Preserving existing data**: Users and testers do not need to uninstall a system to apply a patch. Thus, existing data can be preserved to support continuous operation.
- **Reduced system downtime**: The time needed to apply a patch is usually much shorter than that of a complete installation.
- **Reduced consumption on network bandwidth**: A patch is usually much smaller in size
than an installer. Deploying a patch rather than an installer to a remote destination is faster and more cost effective.

*Increased management cost.* A patch is used to upgrade a particular version of applications. If an application in use is multiple patches away from the target version, these patches must be applied in the right order or else the upgrade may fail. One way to alleviate this problem is to periodically release a service pack and encourage the customers to apply the patch.

*Increased preparation and testing effort.* Preparing a patch can be an error-prone activity. Developers need to figure out the difference between two application versions and include all of these difference parts in a patch. Missing any one of the difference parts could cause the patched application to become incomplete. Such a problem may not be observed until something terribly wrong happens that crashes users’ data. Thus, carefully testing a patch before it is released is vital.

**Known Use:** The *Patch* pattern is widely used for all kinds of software, cross-platform or otherwise. It is used by cross-platform browsers such as Netscape and Firefox [11].

**Related Patterns:** For a major upgrade that may significantly change the structure of an application, producing a *Patch* may be time-consuming and error-prone. In this situation, use an *Installer* instead.

### 4. SOURCE CODE MODULARITY PATTERNS

Source Code Modularity patterns include *Interface Module*, *Platform Independent Module* and *Native Module*. They are used to deal with subdivision of a source code into modules by applying the principles of *separation of concern* and *module decomposition* to cross-platform software.

#### 4.1 Interface Module

**Context:** A cross-platform application contains native code to access platform-dependent services and functions. If the platform-independent parts of the application directly invoke the native code, those parts will no longer be platform independent and the application could be difficult to integrate, test, understand, and extend.

**Problem:** How to isolate platform-specific code from platform independent code?

**Forces:**
- Clear separation of cross-platform application into platform independent parts and native parts is needed to simplify the development, integration, and testing efforts.
- For a better extensibility and modifiability, the platform independent parts and the native parts should not depend on each other.

**Solution:** Define programming interfaces to encapsulate platform dependent ser-
services and to allow a service to have multiple implementations. Put the interface
definition code in Interface Modules and avoid mixing interface definition code with
interface implementation code. This pattern complies with the dependency-inversion
principle that helps organize the dependency relationships between high-level modules
and low-level modules [10]. In the context of cross-platform development, high-level
modules tend to be platform independent while low-level modules tend to be native.

 Creating module dependency relationships through interface makes modules easier
to build on a CI system. For example, ezMonitor needs to read a particular region of
physical memory to get hardware-related information such as the motherboard serial
number and BIOS version. Since the way of reading physical memory is platform de-
pendent, an IPhysicalMemoryOperation interface is defined which contains necessary
operations for ezMonitor’s application logic module to read required information from
physical memory without knowing its underlying implementation; see Fig. 4. The native
implementations for Windows and Linux are developed by two different teams. The
Windows team uses the open source WinIO library which provides a kernel-mode driver and
a user model DLL to simplify the implementation of reading physical memory. The Lin-
ux team uses standard C functions such as open, mmap and munmap to implement the I-
PhysicalMemoryOperation interface to read physical memory by opening the /dev/memfile.

By introducing the Interface project, not only Windows and Linux implementations
but also the platform independent application logic can be independently developed,
built, and tested.

![Diagram](image.png)

**Fig. 4.** The interface IPhysicalMemoryOperation separates platform independent modules from
native modules.

**Resulting Context:** For a software development team that applies incremental and itera-
tive development, it is reasonable for a particular interface to have only one essential
implementation initially and to have additional implementations added later only when
they are really needed. In other words, at the early stage of development, an interface usually has only one implementation. In this situation, an interface and its implementation could be placed in the same project to simplify configurations in a CI system. This is very common when developers use programming languages such as Java and C# that directly support the Interface construct in the language level.

Placing an interface and its implementation in the same project becomes problematic when an additional implementation of the interface is needed. The existing implementation is useless for the new implementation, but being collocated with the interface in the same project, it is included unnecessarily by the new implementation. Thus, even if one implementation for an interface is sufficient initially, it is still a good idea to separate the interface and the first implementation by placing them in two different projects.

**Known Use:** Logan suggests defining abstractions (i.e., interfaces) to hide platform-specific functionality and to foster code sharing across platforms [11]. The Java Database Connectivity (JDBC) defines interfaces without including any implementation, allowing vendor- and platform-specific implementations to be provided [20]. The open source Chromium defines interfaces, for example the RenderWidgetHostView interface located in the content/Source/browser folder, and provides Windows, Linux, and Mac OS X implementations in the chrome/Source/browser folder [21].

**Related Patterns:** A Native Module usually references an Interface Module and provides an implementation to an interface defined in the Interface Module. A Platform Independent Module accesses platform-specific services via an Interface Module.

### 4.2 Platform Independent Module

**Context:** A cross-platform application is composed of both platform independent code and native code. In the monolithic build model, the two types of code are mixed together in a same project. Obviously, native code must be built on each supporting platform. Without separation, the build for platform independent code is duplicated on multiple platforms. As a result, the build takes longer to complete. Developers can become dissuaded from committing code as frequent as possible to avoid long builds. The situation aggravates further if a build is broken, since with less frequent commits and builds, it would be difficult to trace to the causes that broke the build.

**Problem:** How to modularize code for build?

**Forces:**
- You want to avoid long builds.
- You want to keep build scripts conceptually simple.
- You want to simplify build environment provisioning.

**Solution:** Classify implementation code into two types: platform-independent and native. Put platform-independent code in Platform Independent Modules and build them in the CI system’s local environment. For example, you may build all platform-
independent code in a SUSE 11 with JDK 1.6. The resulting platform-independent binary code can be run and tested on Linux-based platforms such as RHEL, CentOS, SUSE, Ubuntu, and so on, and Windows-based platforms such as Windows 7, Windows 2008, Windows XP, Windows 2003, and so on.

By definition, a module is platform-independent if the runtime environment needed by the module is available in all target platforms. However, sometimes the boundary of whether a module is platform-independent is not as clear-cut. For example, suppose you have a Java utility named `FileUtils` that provides functions to recursively copy, delete, and move files as well as folders. `FileUtils` looks like a Platform Independent Module since it uses the standard Java APIs to manipulate files and folders. In reality, `FileUtils` is better treated as a Native Module rather than a Platform Independent Module since there are differences between Windows and Linux file systems, e.g., the maximum lengths of file names allowed. Thus, although the `FileUtils` can be successfully compiled on Windows and Linux platforms, the test result can be different in cross-platform testing.

**Resulting Context:** A Platform Independent Module can be built in the developer’s local working environment. This enables the developer to perform a Local Build before committing a Platform Independent Module and triggering a build, which is usually called a commit build, on the CI system. This reduces the probability of a Platform Independent Module caused broken build. Although a Platform Independent Module should be tested on all target platforms, the tradeoff of testing time for deadline by reducing the number of target testing platforms is relatively safe. The trade-off is not applicable to Native Modules as reducing the number of target testing platforms could raise significant quality issues.

**Known Use:** This pattern is observed in the cross-platform SWT library and the Chrome browser. For example, Eclipse SWT Custom folder Widgets/common/org.eclipse.swt.custom contains platform-independent code [21]. For Chrome, platform-independent code is defined in the folder chrome/Source/browser/tab_contents [21]. Logan [11] proposes a project structure to separately store platform-agnostic code and native code. The platform-agnostic code and the native code are stored in the folder src and the folder src/[platform], respectively, where [platform] is the specific operating system that the native code is intended for.

**Related Patterns:** A Platform Independent Module depends on Interface Modules to invoke services provided by Native Modules. Apply Remote Build to test that a Platform Independent Module can actually run on all supported platforms.

### 4.3 Native Module

**Context:** The application under development contains native code for both Windows and Linux platforms. For Windows platform, two compilation environments are required, including the Microsoft Visual Studio.NET 2008 and the Microsoft Visual C++ 6.0. For Linux platforms, gcc 4.4.3 is required.

Suppose that the CI system runs on SUSE 11 with gcc 4.4.3. This environment can be used to build Linux native code but not Windows native code. An environment suita-
ble for building native code – one that is different from the CI system’s local environment – is required.

**Problem:** How to modularize code for build?

**Forces:**
- Native code can only be built on a supported platform, which may not be the platform that the CI system runs on.
- You use a single CI system that only provides a default build environment.
- Native code should be tested on all supported platforms.

**Solution:** Classify implementation code into two types: platform – independent and native. Put native code in a **Native Module**. In the CI system, be ready to prepare remote build environments for building **Native Modules**. For example, you may build all Linux native code in a SUSE 11 with gcc and build all Windows native code in a Windows XP SP3 with Visual C++ 2008. The resulting Linux binary native code can be run on RHEL, Cent OS, SUSE, Ubuntu, and so on. The resulting Windows binary native code can be run on Windows 7, Windows 2008, Windows XP, Windows 2003, and so on.

**Resulting Context:** **Native Modules** can be built and tested on virtual machines if they are operating system dependent rather than hardware dependent. In this situation, it is easier to prepare, manage, and maintain the build environment. For hardware dependent **Native Modules**, developers should use a **Remote Build** to avoid a broken build before they are committed to a CI system for integration.

**Known Use:** The cross-platform SWT library uses operating system-specific APIs to create native widgets to improve its performance and to provide the nature look and feel on each supported platform [22]. The native code is exported as several .dll files in Windows platforms and .so files in Linux platforms. **Native modules** are also used in Chrome. For example, the process sandboxing is implemented in more than 100 files for Windows platforms; on Linux platforms, sandboxing systems are available although different Linux distributions have different sandboxing APIs [13]. Logan suggests placing platform-specific prebuild and native code in a lib/[platform] and scr/[platform] folder, respectively [11].

**Related Patterns:** A **Native Module** may implement interfaces defined in an **Interface Module**. An **Integration Workflow** may apply a **Local Build** and/or a **Remote Build** to build a **Native Module**.

### 5. BUILD EXECUTION PATTERNS

Build Execution patterns include **Local Build**, **Remote Build**, **Integration Workflow** and **Single Responsible Person**. They are used to deal with workflow of continuous integration and strategies for preventing broken builds.
5.1 Local Build

**Context:** When a developer completes a task, code and tests pertaining to the task are added, and other code can be changed or deleted. A build is needed to make sure that the changed code does not cause a break. The developer can do this in his/her personal development environment or on the CI system.

**Problem:** Which environment to perform a build in?

**Forces:**
- You want to keep the time short to get a fast feedback by doing build after completing a task.
- Your local development environment or CI environment coincides with a target platform of your cross-platform software product.

**Solution:** Perform a build locally. There are two kinds of local environments: the one used by the developer where he/she does the development tasks (which is called *Local Build – Developer*) and the one used on the CI system (which is called *Local Build – CI*). In a *Local Build – Developer*, the developer conducts a build before committing code to the repository by issuing a build command in his IDE, or by running the build script used by the CI system in a clean directory. The latter is especially recommended because doing so removes dependencies on the IDE and therefore reduces the possibility of causing a broken build on the CI system. To avoid a long build, a *Local Build – Developer* makes use of unchanged and pre-built libraries from the *Single Shared Library*.

In a *Local Build – CI*, the CI system automatically conducts a build in its local environment when code is committed or upon request. A *Local Build – CI* can make use of *Single Shared Library* to shorten build time if it is actively used in place of a *Local Build – Developer* by the developers during development. However, make sure to conduct a *Local Build – CI* at least once a day that starts from scratch to guard against outdated components and to update the *Single Shared Library*.

Note that the *Platform Independent Modules* and *Native Modules* whose target native platform happens to be the same can be built either in the local environment or in the CI system. However, *Native Modules* may need to be built elsewhere.

**Resulting Context:** A *Local Build – Developer* normally requires a shorter build time than a *Local Build – CI*. In addition, conducting a *Local Build – Developer* before committing code to the CI system is a common practice and a useful way to prevent a broken build. Although the time needed to conduct a *Local Build* can be significantly reduced by so doing, the probability of causing a broken build can increase since only part of the entire application is rebuilt.

**Known Use:** All known CI systems and build tools support the *Local Build* pattern.

**Related Patterns:** Developer should use a *Remote Build* if his/her development environment is not eligible for conducting a build locally. Apply *Integration Workflow* to plan execution sequence of the build jobs, for example, by planning a *Local Build – CI*
to take place following a successful a Local Build – Developer.

5.2 Remote Build

Context: A cross-platform software product contains platform-dependent code modules for Windows and Linux platforms, respectively. One way to build these modules is to install two CI systems: one on Windows and the other on Linux, and conduct a Local Build on each platform. This solution is acceptable for building projects supporting only a few platforms, but not so if, say, twenty different platforms are to be supported. Installing and maintaining twenty CI systems does not look like a good idea.

Problem: Which environment to perform a build in?

Forces:
• A local environment is eligible for building some but not all the native modules.
• It is too complicated and not cost-effective either to have a CI system conducting a Local Build for each of the Native Module’s target platform.

Solution: Use a single CI system to deploy the build for a native module to remote platforms, perform the build remotely, and collect the remote build results. Each of the remote platforms provides an environment that is eligible to build the project on a target platform. The remote platforms combine to cover all the target platforms.

The Remote Build pattern, also known as the Distributed Build pattern, is supported by many CI systems including Jenkins, JCIS, CruiseControl, Buildbot, Bamboo, and TeamCity [15, 16, 23-26]. For example, suppose Jenkins is used as the CI system and runs on a Windows Server 2008 box in the development of an application that uses named pipes as a means of inter-process communication. Since the implementations of named pipes in Windows and Linux are totally different, two Native Modules for named pipes are required. The Jenkins CI must be configured so that it builds the two Native Projects on both platforms.

To do this, two virtual machines (VMs), one with Windows 7 OS installed and the other with RHEL 6.1 OS installed, are created. Further, the required tools for builds (e.g., compilers, source code analysis tools, and acceptance testing libraries) are installed on the respective VM as well. A build project for the application under development is created on Jenkins; Jenkins remote build agents (slave agents) are deployed to the VMs; and Jenkins build project is configured to use the VMs as remote build environments. Upon a build request, Jenkins responds by dispatching the project to the two VMs, which then execute build tasks and makes the results available to Jenkins. Jenkins then collects the build results and makes them available to the developers on the project’s Web page.

Resulting Context: In addition to be used to run specialized build jobs in specific operating systems, the Remote Build pattern can also be applied to speed up build process by running build jobs in parallel. For example, it may take several hours to completely run your integration test on a single remote build machine. To reduce the time of integration test, you can divide the integration test script into a number of disjoint integration test
suites and configure a build job to run all of the integration test suites in parallel.

For CI engineers, applying the Remote Build pattern is relatively easy because this pattern is directly supported by many CI systems. However, for developers who want to apply this pattern to prevent a broken build before committing code, they may need to find or create suitable tools to do so because the Remote Build is usually not a built-in feature of an IDE. As a result, developers may be discouraged to conduct Remote Build before committing their code. The development team needs to define a rule to deal with this situation: either accepts it or spends some time to establish a common development environment that supports the Remote Build pattern.

Known Use: CI systems such as Jenkins, JCIS, CruiseControl, Buildbot, TeamCity, and Bamboo support this pattern [15, 16, 23-26].

Related Patterns: Remote Build depends on Integration Workflow to dispatch build jobs to remote build environments. If your local environment is eligible for creating a build, you can use a Local Build instead.

5.3 Integration Workflow

Context: A cross-platform software product is structured into a number of interdependent projects. Each project defines certain tasks to be done when it is built. At the early development stage, the build scripts of the projects involve a few basic build tasks such as compiling, testing, and packaging. As developers become more familiar with CI practices, advanced tasks such as code coverage as well as static code analysis and automatic acceptance testing are added.

Being able to build individual projects alone is not enough to produce an installation program for the product. Since the projects are interdependent, a change in one project should cause its dependent projects to be rebuilt to reflect such a change. In addition, if an installation program is built from scratch, the build sequence of the interdependent projects must be clearly defined. Otherwise, the end product (i.e., the installation program) may contain incomplete binary libraries and as a result cannot be executed.

Another issue to be considered is the build time. The time needed to build a project must be short enough so that developers are willing to continuously commit code for build. To reduce the build time, unrelated build tasks such as generating API documents and counting line of code can be executed in parallel. Acceptance test cases that potentially take a relatively longer time to complete can also be divided and run in parallel to further reduce build time. However, doing so unavoidably complicates the workflow of a build script and the configuration of the CI system.

Problem: How do we manage build jobs?

Forces:
• A project contains different types of build jobs such as compilation, testing, and static code analysis.
• Build jobs are usually added to a project incrementally.
Projects must be built according to their dependency relationships. Otherwise, the end product may contain incompatible code and fail to execute.

You want to reduce the build time of a project.

**Solution:** Design integration workflows to control build jobs. Two types of essential integration workflows are concerned: *intra-project workflows* and *inter-project workflows*. The former decides which integration activities (e.g., compilation, testing, test coverage analysis, API document generation, and software packaging) to be included in a build and the execution order of the integration activities. The latter decides the build order of all projects according to the project dependency relationships.

**Resulting Context:** Integration Workflow guides the execution order of build jobs. Usually, build jobs of different kinds can be easily added into an integration workflow incrementally. Thus, even if developers are not familiar with CI practices, they can still define basic build jobs such as compilation and unit testing at the beginning of practicing CI. Later, advanced build jobs such as code coverage analysis and automatic acceptance testing can be added one after another.

Some advanced integration scenarios, for example, running build jobs in parallel to reduce build time and to support cross-platform integration may be difficult to implement in the past. Nowadays, with the help of build tools and CI systems supporting distributed builds, implementing such advanced integration scenarios is no longer a difficult problem. However, the implementation of elaborated Integration Workflows usually requires a deep understanding of particular build tools (e.g., Ant or Make) and CI systems (e.g., Jenkins).

**Known Use:** Almost all non-trivial software products are built by some kind of integration workflows and many tools are available to support this pattern, e.g., a make file is commonly used to define integration workflow for C/C++ programs; Ant is very popular for defining integration workflow for Java programs; Jenkins’ parameterized triggers and multi-configuration build functions combine to allow users to define multiple configuration settings for a build and then choose one to execute at runtime to change its build jobs and workflows [15]; and so on.

**Related Patterns:** An Integration Workflow uses a Local Build or a Remote Build to execute its integration activities. You can also apply Remote Build to concurrently run build jobs.

### 5.4 Single Responsible Person

**Context:** Your software product is developed by a number of developers. Each of the developers can check in code for build and integration. Although your team has tried many practices to prevent a broken build, it still happens sometimes.

When a developer commits a piece of code that immediately causes a broken build, the developer is usually aware of this situation and will try to fix the broken build as
soon as possible. But there are situations that a broken build may be ignored inadvertently. For example, in a project where developers frequently commit code to the repository, a stream of commit builds may take place in a short span of time. If one of the commit builds is broken, the individual developers may think that someone else will take care of the broken build. Consequently, the broken build may remain uncared for.

The case of a broken build that occurs during an overnight build is likely to get ignored as well. Although the project setting can be readily configured to have all developers receive a notification from the CI system when a broken build occurs, there is a good chance that this alert gets ignored because people tend to think that “someone else will fix it”.

**Problem:** Who is responsible for the health of every build?

**Forces:**
- A broken build happens.
- Fixing a broken build is usually a tedious and time-consuming task.

**Solution:** Assign one person in development team to get notified when a build is broken. The *Single Responsible Person* has the responsibility and authority to make sure that a broken build will be fixed as soon as possible. Once a build is broken, the *Single Responsible Person* needs to confirm that the developer who causes the broken build is aware of this problem and has taken recovery measures immediately. The *Single Responsible Person* continuously monitors the recovery progress until the broken build is fixed.

Sometimes it is difficult to find out the person who broke a build, e.g., a broken build caused by overnight build scripts. In this situation, the *Single Responsible Person* is given the authority to “shutdown the production line” (stop all development activities), call a brief meeting and invite all developers to get together to find out the possible causes of the broken build, and then assign a person to fix the broken build.

**Resulting Context:** This pattern prevents broken builds from being ignored, especially the ones that occur in overnight builds. For a small agile team with fewer than ten developers or so, the *Single Responsible Person* may be one of the development team members. For large projects, the role of *Single Responsible Person* can be played by an individual build group.

**Known Use:** Spolsky describes a story about how to choose the *Single Responsible Person* in the Microsoft Excel team [27]. In short, the person who causes a broken build will be the *Single Responsible Person* until another developer causes a new broken build.

**Related Patterns:** This pattern is used to resolve a broken build of the *Installer, Patch, Platform Independent Module, Native Module, and Interface Module*. The *Single Shared Library* may also apply this pattern and assign a developer to upgrade shared third-party libraries and notify all developers of the upgrade.
6. RESULTS OF APPLYING THE PATTERNS IN A REAL-WORLD SOFTWARE

All of the ten patterns were applied in the development of a real world software, called ezMonitor, which is used for monitoring the health conditions of a computer system. In this section, a brief description of the background information of ezMonitor is given, followed by a discussion on the main results of applying the patterns in the ezMonitor project.

6.1 Pattern in Action: ezMonitor

ezMonitor is developed for monitoring the health conditions of a computer system, e.g., CPU temperature, fan speed, and so on. While developed mainly with the Java language, ezMonitor also includes code for drivers written in C/C++ for accessing information collected by various hardware sensors and health chips installed on the motherboard. In the past, vendors of these health chips regularly provided drivers for Windows platforms but rarely did so for Linux platforms. As a result, motherboard vendors, being under-supported on Linux platforms, usually provided health monitoring systems only for Windows platforms. The circumstance has changed in recent years. Nowadays, kernel built-in Linux-based drivers or drivers readily available from vendors are very common. The new circumstance has led to the decision of making ezMonitor cross-platform to support both Windows and Linux-based systems.

The module structure of ezMonitor is designed by applying the general principles of separation of concerns and module decomposition [9]. Following the CI build patterns of Fig. 1, these modules are allocated into a number of projects. At the product level, an Installer project produces an easy-to-use installation program. Creating the Installer depends on various Platform Independent Modules, which include hardware and operating system independent programs such as Web-based user interfaces and application controllers, as well as Native Modules, which include hardware and operating system specific programs such as drivers for health chips and dynamic-link libraries (.dll or .so files) for calling operating system specific functions and reading/writing physical memory as well as I/O ports. Since ezMonitor supports both Windows and Linux, the Platform Independent Modules invoke operating systems specific services implemented in the various Native Modules only through various Interface Modules.

Successful builds of Platform Independent Modules, Native Modules, and Interface Modules generate various artifacts. In the case of a Platform Independent Module and an Interface Module, one jar file containing Java bytecode and the other containing interface declarations in Java source code are generated. In the case of a Native Module, various dynamic-link libraries, drivers, and platform-dependent shell scripts are generated. These artifacts are made available to other dependent projects through a Single Shared Library. The Single Shared Library 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, it is ensured that different projects that share a same library will always use the same version of that library.

Given the CI workflow planned as above, various types of builds can be defined to
take platform characteristics into account. A *Local Build* using the CI system’s local environment is usually suitable for *Platform Independent Modules*. A *Remote Build* is needed for each of the specific platform required by a *Native Module*. Platform-specific *Remote Builds* can run in parallel to reduce build time. Once the build environments are ready, *Integration Workflows* are defined to direct the CI system to carry put the builds in the correct order.

The development team of ezMonitor makes sure that any broken build receives immediate attention by assigning a *Single Responsible Person*. On a broken build, the designated *Single Responsible Person* has the responsibility of figuring out who is responsible for the broken build, informing the responsible party to take action, and ensuring the broken build is fixed as soon as possible. The *Single Responsible Person* also determined the affected scope. Only teams responsible for projects in the affected scope are notified to respond; teams responsible for projects that are outside of the affected scope continue to work without being held up. The *Single Responsible Person* is the only person who is allowed to upgrade the third-party libraries in the *Single Shared Library*.

Lastly, with all dependent projects successfully built and artifacts written to the *Single Shared Library*, the *Installer* project produces an installation program by packaging the necessary files from the *Single Shared Library*. In addition, a *Patch* project produces quick fixes from the build outcomes of two consecutive versions of ezMonitor produced by the *Installer* project. By applying the *Patch*, the user can upgrade to the new version without reinstalling ezMonitor.

### 6.2 The ezMonitor Project Background

The development of ezMonitor began in 2008 by a Scrum team of seven developers. While some architecture work was performed up-front, focusing on delivering features for the Windows platform first in a fast pace to satisfy the initial customers, ezMonitor had initially adopted the monolithic model [11], which evolved into the component model [3] as supports for various Linus platforms were added to the product. The development team adopted two essential XP practices from the very beginning: unit testing and continuous integration [1]. Initially, CruiseControl was used as the CI server.

A build was triggered by the CI server once a piece of code was committed to the code repository, SVN. Initially, the build pipeline contained compilation, testing, and producing installers. Creating the installers from scratch took about three minutes. However, as more production code and test code were added, it grew to the point creating the installers from scratch took about twelve minutes and running all types of test cases (*i.e.*, unit tests, integration tests, system tests, and *etc.*) took more than one hour. The developers responded to the one-hour waiting time by gradually cutting back on running tests on the CI system. Tests, if at all, were run on the developer’s local development environment. At one point, no tests at all were run on the CI system anymore.

Not only did the monolithic model impact the CI practice, it also obstructed the development as ezMonitor grew in scale and complexity. For example, a front-end developer had to check out the back-end modules merely to reference the data access object (DAO) interfaces defined in the server-side modules. It was time to restructure the ezMonitor to streamline the CI as well as the development activities.
6.3 Main Results

We discuss some interesting results in applying the patterns to the agile development of ezMonitor here:

- In August 2011, the ezMonitor contained about 600K lines of code (LOC), including 550K Java LOC and 50K C/C++ LOC. The build time was reduced to two minutes to create an installer from the Single Shared Library. This is a great improvement from the original build time of twelve minutes to create the install from scratch.

- As the ezMonitor was evolved from the monolithic model toward the component model [11], more and more sub-projects were created. Managing these interdependent projects in the build scripts became a complicated and error-prone task because the team’s initial CI system CruiseControl did not support the management of project dependencies very well. Thus, the team decided to replace CruiseControl with Jenkins. Project dependencies can be managed by using the Jenkins downstream and upstream functions. The Platform Independent Module, Native Module, and Interface Module patterns guided us to break the monolithic structure step by step toward the component model. Without the restructuring, it would not have been possible to take advantage of project dependencies management functions provided by the Jenkins. Note that the component model also helped the ezMonitor team to become a better cross-functional Scrum team. The developers could now pick a development task from the Scrum task board and check out only the necessary projects related to the task. In contrast, in the past, the developers needed to check out the entire code base every time and installed all necessary components in their local development environments every time they picked up a development task.

- The number of test cases increased significantly so that it was not possible to run all of them in the normal build pipeline. Inspired by the XP Ten-Minute Build practice [1], the team took a “time-boxing” approach to run some selected unit tests, especially the ones for ensuring that a working installer is created. Since creating an installer from the Single Shared Library took two minutes, the team had about eight minutes to run unit tests. A test suite containing “eight-minute unit testing” was maintained to verify the installer.

- The Integration Workflow pattern was used not only to build production code, but also to execute test code. The team created additional CI projects to run test cases which were not selected by the eight-minute unit testing suite. For example, a system test CI project called “Check All Web Links” was executed once an hour during 10:00 AM to 6:00 PM on work days to verify that all hyperlinks in the Web UI were correct. This project took about 20 minutes to execute on a “single platform.” Since ezMonitor supports multiple platforms, an integration workflow was devised for the “Check All Web Links” project for dispatching the system tests to multiple remote platforms in parallel to reduce the build time. The ezMonitor also depended on the Integration Workflow pattern to build Native Modules in remote build environments.

- Although notifications of broken builds were sent to all developers, a broken build tends to be discovered late or even totally got ignored until a person had to use the result of the build, which makes the root cause of such a broken build is difficult to trace. The Single Responsible Person pattern advised the team to assign a dedicated develop-
er to take care of the health of every build. The CI system sent a broken build notification to the Single Responsible Person who then interrupted the whole team to deal with the broken build. Usually the person who just committed code to the repository would take the responsibility for fixing the broken build. Although the interruption caused by a broken build may be annoying, it successfully served as a quality indicator and reminded the developers that they need to conduct a Local Build before committing code to the repository.

• Since the software product is divided into projects and each of the projects contains its own build script, it is easier to dispatch the projects to different platforms for build. For example, you can use the Jenkins distributed build function [15] to build not only Native Modules, but also Platform Independent Modules (e.g., to compile code and to run test cases in different platforms.)

• Compared to the monolithic model, the component model can reduce the build time because it is easier to implement incremental build in the latter [3]. In addition, the distributed build function provided by CI systems such as Jenkins and JCIS [15, 16] can also significantly reduce the build time by running build jobs in parallel. A quantitative example is given in [16].

• For cross-platform software, the resulting component dependency graph is usually very complicated. By applying the pattern language, the software dependency graph is managed in two levels: the project level and the product level. In the project level, each project takes care of its dependency graph which is documented in its build script. Each project is a black-box from its dependent project’s point of view. The product level dependency graph is addressed by the Installer project and the Patch project.

• Cross-platform software development requires a variety of programming skills. For example, a development team uses, respectively, C and C++ for Native Modules, Java and JavaScript for Platform Independent Modules, and proprietary scripts provided by vendors of installation tools for Installer projects. It may not be possible for all developers in the team to be familiar with all of the skills needed. Even in the case of a cross-functional team (e.g., a Scrum team), applying the pattern language can still have the claimed advantages. During sprint planning, the different projects derived from the pattern language define a boundary for dividing a user story into tasks. In the meantime, it is much clearer for developers to check out the correct projects to implement each of the tasks.

7. 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 well-known Portland Pattern Repository [28] 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 [4, 6]. Continuous integration systems supporting cross-platform software development include [15, 23-26].
The cross-platform project structure suggested by Logan [11] can be implemented by the Interface Module, Platform Independent Module, Native Module, and Single Shared Library patterns presented in this paper. His work provides many useful practices for cross-platform software development. However, Logan’s work is not presented in the pattern format and does not focus on the CI aspect.

Sutherland et al. reported a project of developing a highly complex Integrated Library System (ILS) [28]. ILS is a cross-platform product that runs on various Microsoft Windows as well as Red Hat Linux and Sun Solaris. The project practiced Integrated Scrum with team members located in three countries across two continents [28]. CI, which takes the form of hourly builds from a centralized repository, is identified as one of the best practices that contributed to the project’s hyper-productivity. A build reportedly took 12 minutes (and 30 minutes if database changes are involved) to complete. Although additional details regarding the project structure is not available, the patterns presented in this paper should be applicable.

Muller and Knoll described an approach for cross-platform automated software builds and testing for vision software library that has a layered architecture [29]. In this approach, a build entails compiling and testing the modules in a bottom up manner, proceeding to the next higher layer only if the previous layers are successfully built. At the top layer are the tutorial applications, in which whether testing result is satisfactory is judged by human operator due to the nature of the application domain. The cross-platform problem is solved by virtualization. As the number of supported platforms is large, the testing is time-consuming. Although the term continuous integration is not used, Muller and Knoll’s approach is technically similar to continuous integration. In particular, the authors propose, as part of the future research, to reduce the time duration of a test run by separating platform dependent code modules from platform independent code modules and applying virtualized testing to the platform dependent parts only. The pattern language proposed in this paper should be applicable to achieve their objective.

Holck and Jørgensen considered the relationship between continuous integration and quality assurance based on a case study of two cross-platform open source software, Mozilla and FreeBSD [30]. Due to the open source nature, decentralized continuous integration is adopted so that developers can access and add contributions to the development version. As such, it is necessary to provide mechanisms for ensuring that developers really integrate their contributions properly. For example, in the case of Mozilla, the change committing developers are said to be ‘on the hook’, meaning that they are to be available to work with the Mozilla build team to fix a broken build. Together, the Mozilla build team works with the developers that have committed changes to play the role of the Single Responsible Person. Before committing changes, the very first rule of the Mozilla Portability Guide asks the contributors to test before committing, preferably with TryServer (operating a Buildbot) if testing on multiple platforms [31]. This can be seen as a form of Remote Build in action.

8. CONCLUSION

This paper documents a pattern language comprising ten patterns for agile development teams to effectively practice CI builds for cross-platform software. In agile de-
development, it is very important to budget build time and scope well to ensure that a working software is produced on a regular basis. These patterns are distilled from known uses of builds in existing software and from our experience in building commercial and open-source cross-platform software. An example is given to illustrate the application of these patterns to structuring and organizing source code modules, managing builds, and executing builds. The result shows it benefits the development in many aspects, including reduced build time, decreased probability of broken builds, and simplified project dependency management, among others.

ACKNOWLEDGMENTS

This work is funded by the National Science Council of Taiwan under grant NSC 99-2221-E-027-096-MY3 and by the Ministry of Economic Affairs of Taiwan under grant 100-EC-17-A-02-S1-135. A preliminary result of this work was presented in the AsianPLoP 2011 and we thank our shepherd Dr. Takao Okubo and the participants in the writer’s workshop for their constructive comments. Special thanks go to Dr. Yu Chin Cheng for offering many valuable comments to the presentation and organization of the paper. We also thank Shi-Chun Wang for his participation in preparing the Known Use part in some of the patterns.

REFERENCES

8. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, MA, 1994.

Chin-Yun Hsieh (謝金雲) is an Associate Professor at the Department of Computer Science and Information Engineering of the National Taipei University of Technology, Taiwan. He received his MS and Ph.D. degrees from the University of Mississippi and the University of Oklahoma, respectively, both in Com-
puter Science. His research interests include object-oriented software engineering, pattern languages and software testing. He is a member of the Software Engineering Association of Taiwan (SEAT).

Chien-Tsun Chen (陳建村) is the Founder of Teddysoft Technology, Taiwan, which provides training and coaching services related to agile and lean software development. His research interests include agile software development, exception handling, and pattern languages. He received the PhD degree from the National Taipei University of Technology, Taiwan, in 2008, in Computer Science. Dr. Chen is a member of the Software Engineering Association of Taiwan (SEAT). Contact him at teddy@teddysoft.tw.