Frameworks on the Rise

Kevin Bierhoff
Two Sigma Investments
www.cs.cmu.edu/∼kbierhof

Ciera Jaspan and
Jonathan Aldrich
Carnegie Mellon University
www.cs.cmu.edu/∼cchristo
www.cs.cmu.edu/∼aldrich

George Fairbanks
Rhino Research
rhinoresearch.com

ABSTRACT

Software frameworks have changed significantly since they were described by researchers more than a decade ago. They have entered mainstream use in most domains of software development, and their structure and interaction mechanisms have evolved. This paper provides a revised definition of frameworks and surveys commonly used frameworks to extract two categorizations: a categorization of the different kinds of frameworks found in the wild and a summary of common interaction mechanisms employed by frameworks. Four popular frameworks are described in detail to illustrate these findings. The paper also discusses unique framework challenges and avenues for future research on frameworks.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, languages

Keywords

Interaction mechanism, inversion of control, plugin

1. INTRODUCTION

Software frameworks are in nearly ubiquitous use by software practitioners today. Yet frameworks and their unique challenges have received comparatively little attention from researchers in the past decade. Early research on frameworks was thorough [Johnson, 1992, 1997b,a, Johnson and Foote, 1988, Fontoura et al., 2000, Riehle, 2000], but the reality of software frameworks today is quite different from when they were first described. Not only are frameworks more commonly used for building a diverse range of applications, but the ways in which frameworks interact with plugins have evolved dramatically as well. As a case in point, frameworks no longer rely exclusively on object-oriented *design patterns* [Gamma et al., 1994] but instead use a variety of mechanisms to instantiate and communicate with plugins. Nor are frameworks always *instantiated*, but instead may exist at runtime independently of plugins.

This paper attempts to remedy this situation by describing the state of the practice and identifying new research paths. The paper describes and categorizes the nature, structure, and challenges of many software frameworks in common use today. We hope that this will renew research interest in addressing these challenges, and we hope that practitioners can gain a better understanding of the "beast" they are dealing with day-to-day.

In this paper we describe four popular frameworks in depth – ASP.NET, Open|SpeedShop, Eclipse SWT, and the Facebook API – because they are good examples of what frameworks do today. We provide a partial guide to categorizing frameworks: whether they have a runtime existence separate from plugins, whether the plugin has initial control, whether they are quality attribute frameworks that hoist management of a particular quality attribute, and whether they are domain frameworks that define concepts and relationships in a specialized domain. We list and organize the *interaction mechanisms* between frameworks and plugins, including callbacks, plugin loading, dependency injection, advice, and framework libraries. As we will see, these are largely independent from the programming language used to implement frameworks and their plugins.

Recent research has aimed at dealing with various framework challenges, such as expression of common interactions with frameworks [Fairbanks, 2007], expression of framework constraints [Jaspan and Aldrich, 2009, Hou et al., 2004], and automatically extracting plugin-framework interactions [Heydarnoori, 2009]. However, we believe that this is only the beginning, and we hope that research will catch up to the remaining challenges of using software frameworks.

2. FRAMEWORKS: A DEFINITION

This paper does *not* provide an *a priori* definition of what makes a piece of software into a framework. Our approach is pragmatic and attempts to crystallize important aspects of framework practice today. It is guided by an examination of a variety of frameworks including the ones listed in Table 1. The interaction mechanisms we describe are important concepts employed by many frameworks and cover the majority of what is found in practice. However, we expect there are additional mechanisms present in frameworks that we have not examined and anticipate the definition to change as frameworks evolve and new frameworks are invented.

2.1 What Frameworks Are

A *software framework*, or simply *framework*, encapsulates architectural code reuse: it provides an architectural skeleton suitable for a particular class of software applications [Johnson, 1997b]. Such a framework is meant to be enriched with *framework plugins*, also called *framework extensions*, to form an application.

Plugins accept the framework's architecture as their own and

receive any quality attributes, such as performance, reliability, or scalability, that follow from that architecture. In addition, frameworks may provide domain-specific concepts and relationships, often as a set of classes and associations, which plugin developers can reuse and extend. The architecture and code reuse provided by frameworks can thus make development quicker and less error-prone for applications in the target domain. Furthermore, the common architecture provided by a framework helps to reduce architectural mismatch [Garlan et al., 1995] between independent framework plugins.

In order to provide architectural reuse, a framework controls not just application structure but also the flow of execution. The framework supports customization by invoking plugin code at defined points during application execution. This *inversion of control* [Johnson, 1997b] creates challenges because developers must know under what circumstances their plugin code will be invoked. Furthermore, in implementing plugin code developers must follow the architectural constraints specified by the framework, so that they will realize the qualities encapsulated in the framework architecture. This can be difficult, especially when the framework constraints are not clearly expressed [Jaspan and Aldrich, 2009, Hou and Hoover, 2001].

2.2 Frameworks Are Not Libraries

The architectural reuse provided by frameworks is different than library-based reuse. First, libraries are typically called from a developer's code, but this situation is reversed with frameworks, which typically invoke the plugin developer's code at times defined by the framework. Once a framework has passed control to a plugin, however, the plugin often has access to library-like functionality provided by the framework.

Second, a library provides functionality suitable for a sub-domain of an application, while a framework defines the domain for the entire application. An accounting application and a graphics application can both use a math library, but only a web application can use a web framework. Using a framework for an application in a mismatched domain is sometimes possible, but awkward.

3. FRAMEWORKS IN PRACTICE

The term *framework* was originally defined by Johnson as "a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact" [Johnson, 1997b]. However, we believe that this definition does not capture the breadth of how frameworks are currently used. Therefore, we will begin by examining four industrial frameworks from a variety of domains, communities, and languages.[1] For each example, we describe how it employs some of the interaction mechanisms we will describe in the section on Interaction Mechanisms.

3.1 Example: ASP.NET

ASP.NET has become one of the largest platforms for commercial web application development and supports one of the largest communities of users; the official ASP.NET help forum alone has over 440,000 registered users alone and thousands of new posts per day. ASP.NET's success is due in part to the fact that developers may gradually increase their abilities within the framework. While their first website may have just a few buttons and textboxes, ASP.NET also supports highly sophisticated functionality for commercial websites.

ASP.NET itself is simply a server which, when it receives a request for a web page, interprets the developer's code to generate that web page and sends HTML back to the client. The developer provides two program artifacts that represent a single web page. The *ASPX file* provides the "view" of the web page. It is similar to HTML, but it contains ASP tags that the framework will interpret into HTML. The *code-behind file*, written in either VB.NET or C#, provides the "model" for the web page. This file contains event handlers that will be called when the user performs some action on the page, and it contains "lifecycle methods" that will allow the developer to insert functionality when the framework is translating the ASPX file into HTML.

Collectively, this pair of files is referred to as a *page* in ASP.NET, and it can be seen as a plugin to the framework. By creating several of these plugins and connecting them together with traditional web links, the developer can create a complete web application. A sampling of the framework-based features of ASP.NET are listed below:

Plugin loading. ASP.NET loads a plugin *dynamically* when it receives a request for a URL. The framework takes the URL and may do some URL rewriting to generate the path to the ASPX file. The framework then loads the code-behind file and begins the process of translating the ASPX file into HTML.

Callbacks. While the framework is translating the ASPX file into HTML, it calls into the code-behind file at pre-defined execution points that represent some interesting phase in the translation process. There are more than 10 of these phases, ranging from "before initialization" to "all web controls loaded in memory with user data" to "just before rendering into HTML" to "just before destroying all memory". These phases are collectively known as the page's *lifecycle.* The plugin may choose to implement methods in the code-behind file that represent each of these phases, also known as *lifecycle events*. This allows the developer to add functionality such as dynamically adding controls to the rendered HTML based on user input in another control, populating a control with database values, or checking for errors in the user input and generating warnings.

Advice. Web pages for a web application share a lot of code, particularly in the look of the web page. To assist developers and prevent code duplication, ASP.NET allows developers to define templates, called master pages, which encapsulate the look of a page. These templates are easily reused by plugin developers by simply declaring which master page the plugin is using. The framework then handles merging these pages before the lifecycle events occur.

In addition to web pages, developers can create new web controls. These are another type of plugin into the ASP.NET framework, also with a view (ASCX files) and a code-behind file. Like pages, they have a lifecycle that allows developers to dynamically change how the ASCX file is translated into HTML.

3.2 Example: Open|SpeedShop

Frameworks are not unique to the Java community; they have existed in some form for decades and can be found in other languages as well. One such example is Open|SpeedShop, a C++-based performance analysis framework for large scale platforms [Open|SpeedShop, 2009]. Using Open|SpeedShop, developers can create new performance analyses while reusing many of the existing tools for dynamic binary instrumentation, data storage, offline analysis, and visualization of results.

Callbacks. Open|SpeedShop has three types of plugins: collectors, views, and wizards. A collector plugin describes how to

[1] Almost all the frameworks described in this paper are written in class-based object-oriented languages, but we do not believe that software frameworks preclude other languages. The wide interest in frameworks in class-based OO-languages is an artifact of the culture of this group, rather than a technological constraint.

instrument the code and gather data. This data may be analyzed directly, or it may be stored in a database to analyze later. View plugins perform any required analysis and display the results to the user, either through the command line or through Open|SpeedShop's GUI framework. Finally, a wizard plugin helps the user configure analysis-specific parameters. Each of these plugins has a separate interface that it must implement; this allows the plugins to be developed independently and reused to create new analyses.

Plugin loading. Open|SpeedShop loads plugins dynamically by looking into a named directory and linking plugins into the framework. The plugins use a naming convention to allow them to be found by the framework.

Framework library. Open|SpeedShop provides access to several common infrastructures, including a dynamic instrumentation tool, a data storage library, and a multicast communication network.[2]

3.3 Example: Eclipse

Researchers and practitioners have, over the past several years, built a large number of plugins that extend Eclipse [Gamma and Beck, 2003]. The authors all developed plugins to Eclipse's Java IDE that make their research available to mainstream Java developers [Fairbanks, 2007, Jaspan and Aldrich, 2009, Bierhoff et al., 2009]. Eclipse is a successful and highly extensible framework; in truth it is more accurately a collection of frameworks that themselves are plugins to other, lower-level frameworks inside the Eclipse platform.

Eclipse is based on a module system that controls module-to-module code visibility and access. On top of that sits a GUI framework, the SWT, that provides functionality similar to X windows, Swing, or the .NET GUI toolkit. Arbitrary GUI applications can be built using the SWT. Since IDEs are GUI applications, Eclipse then layers plugins onto the SWT that facilitate IDE development. The Java Development Tools (JDT) form one of many IDEs built with Eclipse; other languages, including C and PHP, are also available. Some of Eclipse's framework features are the following:

Runtime existence. Various pre-packaged collections of Eclipse plugins can be downloaded as fully functional applications. Eclipse for Java developers, for instance, is fully functional as an IDE, but plugins can be added as needed. Eclipse comes with its own startup executable; plugin writers do not have to instantiate Eclipse or even their own plugin themselves.

Plugin loading. Plugins are added to Eclipse by placing their binaries into a well-known directory. Eclipse reads plugin-provided configuration files that define how this plugin extends other plugins. The existing plugins are in fact responsible for handling these various extensions the new plugin makes to them and for instance display additional menu items and call the appropriate code in the new plugin when these items are selected.

Advice. One of Eclipse's more unique features amongst GUI toolkits is that its module system prevents plugins from accessing code in other plugins unless these other plugins permit the access. Exerting this level of control involves the use of custom "classloaders" that intercept object instantiations and perform security checks around them.

Framework libraries. Eclipse's frameworks provide large libraries of re-usable functionality. For instance, the SWT framework offers dialog widgets, and the JDT provides access to its compiler and typechecker. These are usually accessed either through singletons or direct instantiation (again, with access checks) similar to a conventional library. However, these "libraries" also act as frameworks in notifying their clients of events: dialog widgets propagate events such as mouse clicks to listeners, and the JDT can notify other plugins of changing source code and other events.

[2]While these infrastructures are currently built in to Open|SpeedShop, the team is investigating a redesign that would allow these infrastructures to be swapped out in the same style as the existing plugins.

3.4 Example: Facebook

The Facebook API [Facebook, 2009] has recently received a lot of attention due to Facebook's popularity. It allows separate websites, referred to as "Facebook applications", to access Facebook's social networking data and interact with Facebook users. The Facebook API is a framework because applications can "plug into" and appear on Facebook's websites. Facebook users can find applications in a registry, links to applications can appear on Facebook websites, and in certain cases Facebook will even request content from applications by "calling" application code and displaying the result (for instance, to generate application-specific message attachments). Hence we can think of applications as plugins and the Facebook website as the framework.

The Facebook API stands out from most frameworks discussed in this paper because no static or dynamic linking of code is necessary for Facebook and its applications to interact. Applications are hosted on separate Web servers and can be written in different programming languages than the PHP-based Facebook website. Nonetheless, many framework concepts directly carry over:

Callbacks. Facebook provides websites where application developers can register their applications. They will, in particular, provide the HTTP URLs of entry points into various parts of their application. A required URL is the "canvas page", the welcome page for users. But more URLs can be specified, for instance if the application wishes to be notified when users un-subscribe from it.

Facebook "invokes" applications at various points, by sending HTTP requests to the application-provided URLs. For instance, if a user clicks on a link to an application then Facebook will forward the user to the application's canvas page. When a user unsubscribes from the application, Facebook does *not* forward the user to the application, but it nonetheless sends an HTTP request to the "unsubscribe" URL (if provided). These requests have a documented shape: Facebook promises to include certain parameters in the request, such as the user on whose behalf the invocation is made.

Advice. Facebook (optionally) "wraps" output from the application into Facebook's own look and feel. This is true for application content displayed on Facebook's websites (such as message attachments), but also for application websites such as the canvas page. When content is to be displayed on Facebook's websites then applications have to format their response in certain ways. Application-specific websites, on the other hand, can contain arbitrary HTML.

Framework library. Additionally, Facebook offers a library of functions to applications, which can be used to, for instance, retrieve information about users and their friends. Facebook even defines its own database query language for applications to use when the library functions are insufficient. It is this (traditional) part of the Facebook API that raises privacy concerns because applications may store the information they retrieve, although doing so is expressly forbidden by Facebook.

Constraints. Therefore, Facebook also imposes constraints on applications: they are required to respect users' privacy, and in order to maintain isolation between application and Facebook data, Facebook disables certain HTML and JavaScript features. Obviously, applications must also format their HTTP responses to Facebook in a certain way.

The Facebook API offers extremely loose coupling between plugin and frameworks: applications can in principle be written in any

programming language, as long as they produce the expected HTTP responses. Even the library, while written by Facebook mostly for PHP-based applications, can be invoked from other languages, because it uses HTTP to communicate with Facebook's servers as well. As such, we believe that the Facebook API is an interesting, albeit atypical, example of a framework.

4. CATEGORIZING FRAMEWORKS

Table 1 contains an archival analysis of frameworks used in industry and some of their selected characteristics. A glance at the table, itself only a sampling of existing frameworks, reveals great diversity, so it is worthwhile to identify a few salient overlapping categories of frameworks to help organize and understand how they vary.

Runtime existence. While the earliest frameworks had no runtime existence until the plugin instantiated them, it is now common for a framework to have runtime existence independent of any plugins. Music playing programs can play back music but many of them also allow plugins to extend their core functions, providing support for new media files or internet streams. The Eclipse IDE runs standalone but can also be extended with plugins. GUI frameworks, however, have no runtime existence until a program is written that uses them.

Initialization. In some frameworks the plugin gets initial control, often through a main routine, that performs some initialization then hands over control to the framework. The alternative is for the framework to own the main routine and instantiate the plugins. The secton below on plugin loading describes the mechanics of this in more detail.

Quality attribute frameworks. Quality attributes are extra-functional properties of a system, such as performance, security, or scalability, and they are strongly influenced by a system's architecture. Since a framework can define part or all of an application's architecture, it can *hoist* [Fairbanks, 2010] the management of a quality attribute, taking it out of the hands of the application. EJB is an application server that hoists scalability and concurrency, largely freeing plugin writers from worrying how to achieve these.

Domain frameworks. Domain frameworks provide a collection of objects that work together to define how a domain works. Some domain frameworks encode business domains, like the IBM San Francisco framework [Bunting, 2000], which encodes standard IT domains including accounts receivable and payable. Other domain frameworks encode programming domains, like GUI frameworks, which encode the domain entities in user interfaces including menus and buttons.

5. INTERACTION MECHANISMS

This section catalogues mechanisms that frameworks use to interact with plugins: *callbacks*, *plugin loading*, *dependency injection*, *advice*, and *framework libraries*. We find all of these mechanisms in a wide variety of frameworks, realized with different techniques and programming languages. Common to many of them is the *inversion of control* principle, which shifts control over some aspect of the plugin's runtime behavior to the framework.

5.1 Callbacks

Callbacks are methods implemented by a plugin that the framework will invoke at certain well-known moments [Fontoura et al., 2001]. The framework defines which callbacks a plugin can react to, and the set of callbacks defined by a framework is somewhat dependent on the framework's purpose. The set of callbacks, regardless of mechanism, is always defined by the framework and reacted to by the plugin.

The callback interaction mechanism has been called the *Hollywood principle* ("don't call us, we'll call you") and has been viewed as the defining mechanism of a framework [Fowler, 2005] because it embodies the *inversion of control*. That is, when a program and a library interact, the program is in control and tells the library what to do and when to do it. This is reversed in frameworks, with the framework telling the plugin what to do and when to do it.

5.1.1 Types of callbacks

Typical types of callbacks include:

Lifecycle. Framework-relevant points in the plugin's "lifecycle", such as initialization and destruction (most frameworks).

Events. Events, such as mouse clicks, received from a "user" or another entity in the environment (GUI frameworks).

Hooks. Points in the framework's execution where plugins can contribute to the framework's built-in behavior, such as when the screen is about to be re-drawn or when data structures are populated from an HTTP request (see ASP.NET). [3]

5.1.2 Callback mechanisms

Implementation strategies for this mechanism include the following:

Type-based. Extensions must implement a framework-provided interface or subclass a framework-provided class in which the signatures of callback methods are defined. It is this interaction mechanism that, in object-oriented languages, is often realized with a variety of design patterns [Gamma et al., 1994] such as the template method and observer patterns. Observers, for instance, which are objects implementing an interface for receiving event notifications, can be registered to receive events such as mouse clicks. Template methods define an algorithm skeleton (for instance, for handling an HTTP request) that invoke abstract methods (to be implemented by the plugin) to fill in details at various points in the algorithm's execution. They are popular for implementing framework lifecycle callbacks.

Reflection-based. Extensions place framework-defined annotations on their methods or use a naming convention to mark methods as callbacks. Registering explicitly with the framework is typically not necessary in this case. The JUnit framework, for instance, treats methods whose names start with "test" or are marked with a "Test" annotation as unit tests.

Event loop. Some frameworks, such as Microsoft Foundation Classes, require plugins to have a single event handling loop. Events are often assigned a constant integer value.

Function pointers. In languages like C, a table of function pointers can serve the same purpose as implementing an interface. The Facebook API even uses URLs to define callbacks. Facebook sends requests to these URLs at the right moment. Facebook includes HTTP parameters, which are similar to method parameters, in these requests that indicate the user making the request, etc.

Build tricks. Extensions can provide implementations for predefined function signatures that will be invoked in the right places after the linker connects the framework and plugin code. In this scenario it is harder for multiple plugin points to receive the same lifecycle events. For instance, Apache can be compiled to use a number of different event processing loops with different strategies for distributing requests among threads.

[3] It can be hard to distinguish events from framework lifecycle callbacks because one can cause the other. For example, data structures are populated (a framework lifecycle callback) because a HTTP request was sent by a user (an event).

Framework	Language	Independent runtime existence	QA or Domain framework	Initialization[a]	Plugin loading	Callbacks	Dependency injection	Advice	Key-value coding[b]	Framework library
ASP.NET	C#, VB.NET	No	Domain	Framework	Naming convention	Hooks	Framework objects	Plugin-defined	Generic and code-gen.	Reuse
OpenSpeedShop	C++	No	Domain	Framework	Naming	Lifecycle	No	Both	No	Reuse
Eclipse	Java	Yes	Domain	Framework	Configuration files	Events, lifecycle	No	Framework-defined	No	Registration, reuse
Facebook	Any (PHP)	Yes	Domain	Both	Configuration (webform)	Events	No	Framework-defined	No	Framework control, reuse
Applets	Java	No	Domain	Plugin	Programmatic	Events, lifecycle	No	No	Generic access	Registration, reuse
Apache	C	Yes	Both	Framework	Configuration files	Hooks	No	Framework-defined	No	Reuse
AWT / Swing	Java	No	Domain	Plugin	Programmatic	Events, hooks	No	No	No	Registration, reuse
Corba	Various	Yes	Both	Both	Programmatic	Lifecycle	No	Framework-defined	No	No
Enterprise JavaBeans (EJB)	Java	Yes	Both	Framework	Annotations, config files	Lifecycle	Framework objects, OR-mapping	Both	No	Framework control
JUnit / NUnit / ...	Various	No	Domain	Framework	Annotations, interfaces	Lifecycle	No	Both	No	Reuse
Microsoft Foundation Classes (MFC)	C/C++	No	Domain	Plugin	Programmatic	Events	No	No	No	Registration, reuse
.NET GUIs	C#	No	Domain	Plugin	Programmatic	Events, lifecycle	Framework objects	No	Static code generation	Registration, reuse
Ruby on Rails	Ruby	No	Domain	Framework	Naming	Events	OR-mapping	No	Dynamic code generation	Reuse
Servlets	Java	No	Domain	Framework	Config files	Events	No	No	Generic access	Reuse
Spring	Java	No	Domain	Depends	Config Files	Lifecyle, events	Component wiring	Plugin-defined	Generic access	Framework control, reuse
WebObjects	Java	No	Domain	Framework	Config files	Hooks	OR-mapping	No	Generic and code-gen.	Reuse
X Server	C	Yes	Domain	Both	Programmatic	Events	No	No	No	Registration, reuse

Table 1: An archival analysis of frameworks and their characteristics

[a] *Both* means framework and plugin run on separate machines or processes.
[b] Not currently discussed in the text. We are investigating this as an additional mechanism.

5.1.3 Callback variation points

Optionality. Some callbacks are optional; others are required. One way of thinking about a required callback is that a plugin is not a valid plugin unless it is able to handle the callback. In practice, however, plugins often do nothing during a required callback, and therefore frameworks have moved to using optional callbacks. For instance, the EJB 2.0 specification defined various interfaces that all plugins had to implement. In EJB 3.0, the callbacks previously required by these interfaces are now marked with annotations. If a plugin does not use a particular annotation, then it wishes not to do anything during that callback.

Registration. Plugins sometimes have to explicitly register to receive a particular callback; at other times the mere presence of a callback method will prompt the framework to invoke that method at the appropriate time. Plugins typically have to explicitly register to receive events, while lifecycle callbacks are often delivered automatically.

5.2 Plugin Loading

When a framework starts, it must somehow discover the plugins, and these plugins must be instantiated. Ralph Johnson's original paper on frameworks [Johnson, 1992] describes how, in Smalltalk, the framework was told about plugins by modifying a dictionary in the Smalltalk image. This mechanism is not commonly used now because no mainstream languages have an equivalent to the Smalltalk image. Instead, either the plugin adds itself to the framework, or it leaves declarative clues for the framework to find it. Examples of these clues include configuration files, annotations, naming conventions, and placement in a known directory.

Either frameworks or plugins can own the main routine, but in both cases the framework eventually takes over control flow. When a plugin has the main routine, developers have more control over the startup process, but this option accommodates only a single plugin. When the framework has the main routine, they can be independent functional applications (Eclipse, for instance).

Programmatic. Plugins can be run directly and subsequently register themselves with the framework. With this choice, the framework is sometimes called a *toolkit*. The style of use is often that the plugin has main routine, does some initialization (e.g., creating a window, adding widgets to it) then hands control over to the framework a single time and subsequently only gets callbacks. The AWT framework, for instance, relies on the application developer to instantiate windows and dialogs as needed.

Declarative. Extensions to be loaded can be defined in a declarative way. At startup time, the framework reads these declarations and is responsible for instantiating, typically using reflection (see below for details). Declarations can take the form of configuration files or markers.

With configuration files, the framework instantiates plugins based on entries in declarative configuration files. Eclipse, for instance, expects archives with plugin binaries (JAR files) to be placed into a particular directory and these JAR files to contain an XML configuration file. Eclipse then uses reflection to instantiate classes mentioned in the configuration file.

With markers, the framework examines available classes and instantiates marked components. The markers vary, and can be annotations, implemented interfaces, or class/method naming conventions. For example, classes to be loaded as EJB plugins can be marked with framework-defined annotations. The EJB framework searches known directories for classes with these annotations and instantiates them as needed. Many EJB frameworks can even discover and instantiate "new" and replaced plugins while running.

Some frameworks that support declarative loading (such as Eclipse) also support plugin-initiated loading in order to allow plugins to instantiate new framework plugins at runtime.

5.3 Dependency Injection

Dependency injection is a mechanism found in many recent frameworks that allows the frameworks to instantiate any resources that a plugin needs on the plugin's behalf. This idea complements the idea of callbacks by allowing plugins to indicate their need for a particular resource, such as a database connection, or a reference to another plugin. At the appropriate time, the framework will provide that resource to the plugin: the framework *injects* the requested resource into the plugin.

5.3.1 Types of Dependency Injection

Dependency injection seems to come in a number of different flavors that are useful to distinguish. Notice that in all cases, the desire is to avoid plugins having to set up dependencies to other components themselves. Thus, dependency injection is inversion of *dependency* control away from the plugin and towards the framework (somewhat similar to a linker, but object- rather than type-based).

Injecting framework objects. Objects created by the framework can be injected to give plugins access to them. For instance, EJB plugins are injected with a "context" object that allows them to call framework methods. The context object implements a published interface, but its implementation is framework vendor-specific. Thus, this form of dependency injection promotes the framework's ability to instantiate classes of its choosing.

Component wiring. References to plugin-defined components may be injected into each other to establish the "wiring" between these components. Doing so makes this wiring more flexible and simplifies, for instance, inserting an intermediary component somewhere or creating circular dependencies. The Spring framework for instance can instantiate a set of plugin-defined objects ("beans") and inject references to beans into other beans based on a declarative configuration. Ideally, individual beans do not instantiate any objects themselves. This is often implemented using key-value coding.

Object-relational mappings. A variety of frameworks for handling object-relational mappings were developed over the last decade, including EJB, Hibernate, Apple WebObjects, .NET persistency, and Ruby on Rails. They all provide support for representing relational databases with objects. In these frameworks, the classes representing individual database tables are often defined in framework plugins, but they are instantiated and populated (with cell values from the database) by the framework, and references to other tables are typically injected into these objects as well.

5.3.2 Implementation Strategies

Well-known methods (interface implementation). Plugins can implement interfaces that define methods for receiving certain resources. Similarly to a callback, the framework will instantiate the resource and invoke the plugin's methods (typically once when the plugin is initialized) to provide the resource.

Dependency annotations. Plugins can annotate methods or even fields as receivers of resources, and the framework will use reflection to assign the resource to the annotated field or invoke the annotated method.

Configuration file. Some frameworks such as the above mentioned Spring framework can inject object references into other objects according to a declarative configuration file.

Auto-generated code. Frameworks can generate code that initializes the resource. For instance, the .NET GUI framework generates code that instantiates and configures GUI widgets in dialogs

according to what the developer configures graphically with a special "form editor". Fields holding the various form elements, such as checkboxes, "magically" become available in the developer-written code.

5.4 Advice, Wrappers, and Interceptors

Advice is a term used in Aspect-Oriented Programming (AOP) to describe code executed "before"', "after"', or "around" a particular operation. Advice on a method can be seen as intercepting calls to the method and executing the advice code before and/or after the "advised" method is invoked. The framework controls the order that the advice occurs in, while the individual pieces (both the advice and the thing being advised) are typically unaware of or at least oblivious to what is occurring. Thus, advice represents inversion of *execution order* control.

5.4.1 Types of Advice

Frameworks sometimes use advice in their implementation, allow plugins to define advice, or both.

Implemented by framework. Advice is often implemented by the framework to "guard" plugin actions. EJB framework implementations use advice to check whether the user invoking an operation on an EJB plugin is allowed to do so: they advise the invoked operation with the necessary check. As another example, the Facebook API defines a custom mark-up language, FBML, for conveniently inserting Facebook resources (such as a user profile picture) into web pages generated by a Facebook application. This process gives applications a look-and-feel that is similar to Facebook's own websites. This re-writing of FBML into HTML is the Facebook API on behalf of Facebook applications and can be seen as "after" advice.

Implemented by plugins. Advice has become so important that EJB plugins themselves can, since EJB 3.0, specify methods they want invoked "before" or "after" their other operations – for instance, to perform custom access control. As another example, OpenSpeedShop advises a target program to measure its performance, and it allows plugin developers to write performance-related advice. The framework takes care of invoking the plugin-defined advice when executing the target program. In these cases, the plugins define the advice, although the framework still takes care of applying it.

5.4.2 Implementation Strategies

Though using an AOP programming language such as AspectJ directly allows the programmer to create advice, it also involves using a special compiler and language constructs. In practice, frameworks are not written in an AOP language and instead implement the needed mechanisms in a conventional programming language:

Weaving. In true AOP fashion, advice can be placed by rewriting source code or bytecode. This is not an option if such artifacts are not available. Since weaving requires a compiler-like infrastructure, this option is unpopular.

Wrappers. A wrapper is a framework-generated class with the same interface as the advised class. It executes the desired advice before and/or after calling methods of the wrapped class. The hard part, of course, is to make sure that the rest of the program calls the wrapper, rather than the wrapped class directly. This sometimes requires plugins to not instantiate other plugins directly but rather to use a framework-provided mechanism for doing so.

Advice-aware programming. Code can be written to allow advice, for instance by explicitly calling the framework to perform any advice. As case in point, Facebook applications have to include Facebook-provided code that triggers the replacement of FBML elements with suitable HTML and JavaScript before sending the result to the user's Web browser. By not including the Facebook-provided code, Facebook applications can communicate with their users directly. Obviously, this approach is error-prone because plugin developers may simply forget to invoke the advice they need.

5.5 Framework Libraries

Frequently, frameworks not only call *into* plugins, but they also provide services that plugins can invoke. Notice that this often implies the possibility of reentrant calls in the framework, since the plugin will call the framework as part of a callback that the framework invoked.

5.5.1 Types of framework-provided libraries

Providing such a "library" to plugins can have several motivations:

Controlling framework services. Framework plugins gain influence on how the framework does its work. For instance, EJBs can start and stop database transactions by calling appropriate methods on the frameworks. (Doing so is optional; alternatively, transaction demarcation can be configured declaratively with annotations or XML files.)

Component registration. Framework plugins are able to programmatically register new components with the framework. GUI frameworks such as AWT or even Eclipse's SWT, for instance, allow plugins to create new dialogs and register event listeners with the framework. Doings so puts the registered components under the framework's control and the framework will call them back as desired.

Re-use. Framework plugins can re-use implementations of domain-typical operations. For example, the Facebook API provides various functions that allow Facebook applications to query information about the current user and her friends. It also includes a simple AJAX (Asynchronous Java and XML) implementation. Both of these are very useful in creating social networking websites.

5.5.2 Implementation mechanisms

Implementation strategies for this interaction mechanisms are surprisingly manyfold, considering this mechanism is very similar to conventional libraries.

Direct instantiation. Conventional libraries are usually implemented as classes that clients can instantiate. Direct instantiation of framework classes by framework plugins is sometimes allowed, in particular in GUI frameworks where plugins can instantiate widget classes directly.

Callback interfaces. In order to avoid dependency on framework *classes*, the framework frequently "injects" (using dependency injection) a object implementing a well-known *interface* into the plugin. This object provides framework services.

Singleton objects. Similar to callback interfaces, plugins are sometimes expected to use certain *singletons* [Gamma et al., 1994] to invoke framework services. This is for instance popular in Eclipse.

Configuration-based instantiation. Frameworks that need to expose complex data structures to plugins can instantiate the data structures on behalf of the plugin and use dependency injection to provide them to the plugin. ASP.NET, for instance, instantiates objects representing Web forms based on plugin-defined ASPX files.

6. DISCUSSION

6.1 Differences Due to Programming Languages

Frameworks are often some of the more creative examples of programs written in a particular language, taking advantage of es-

oteric or "dangerous" features (such as reflection) that most programmers try to stay away from. Partially because frameworks take advantage of these features, there is no question that framework implementations depend enormously on the programming language they are written in. C-based frameworks, for instance, have a hard time with object-oriented design patterns, although many patterns can certainly be "hacked up". Reflection is very popular in the framework writer's toolbox, and so languages without reflection will require other solutions. Finally, the ability to add classes and methods at runtime makes dynamically typed languages attractive for framework development. Since frameworks operate on the fringes of the programming language they are written in we believe that they are a very good starting point for understanding the strengths and weaknesses of different languages.

Differences in how interaction mechanisms are implemented are therefore partially due to which programming language is employed in each mechanism, but also due to the environment that the framework operates in (for instance, remote vs. local invocation of plugins). In addition, there seem to be mechanisms that are simply "in style": a few years ago, XML-based configuration was the state-of-the-art in many frameworks. Annotations have since been introduced into new and existing frameworks, such as EJB (previously XML-based) and JUnit (previously used naming convention). However, this paper shows that these different mechanisms serve a handful of purposes. Framework implementations may appear to be different, but they have a lot of common goals.

We believe that interaction mechanisms put a new twist on many of the fundamental concerns of a programming language that researchers have identified over the last several decades, such as modularity, low coupling, and re-use: frameworks certainly achieve very low coupling between highly modularized components and enable substantial re-use of complicated machinery. But they do so by resorting to mechanisms that are often considered "unsafe", such as reflection or even a dynamically typed programming language. It remains to be seen whether interaction mechanisms can be supported by programming languages more directly.

6.2 A Common Vocabulary

The interaction mechanisms we identify in this paper could form a common vocabulary for talking about the mechanics of frameworks. Frameworks have been characterized as "components + patterns" [Johnson, 1997b], and this paper shows that object-oriented design patterns can indeed be used to *implement* some of the interaction mechanisms we have identified. But nowadays many interaction mechanisms are typically implemented using reflection or other "unsafe" mechanisms rather than design patterns. Thus, interaction mechanisms capture a framework-related purpose, while the use of design patterns is a technical choice for implementing the mechanisms.

Our discussion of implementation mechanisms reveals that the same mechanisms (in particular, design patterns, configuration files, interfaces, and annotations) are used for different interaction mechanisms. Moreover, the same interaction mechanisms come into play in frameworks for vastly different purposes: domain-specific as well as quality attribute frameworks; frameworks for developing desktop applications (such as Eclipse) as well as for highly distributed applications (such as Facebook). Many of the differences between frameworks appear to be motivated by the specific concerns and constraints that the framework addresses, but they still use similar interaction mechanisms to do so. Therefore, we believe that interaction mechanisms provide a level of abstraction suitable for documenting and discussing frameworks: independent from implementation specifics, and independent of the framework's actual purpose.

6.3 Remaining Challenges

Understanding non-local framework-plugin interactions. In order to accomplish a single task, a plugin must interact with various parts of a framework, perhaps implementing several callbacks across several subclasses. One empirical evaluation showed that 15% of framework-plugin interactions served multiple goals [Fairbanks, 2007]. Identifying a set of interactions as related is exacerbated by the fact that many framework interaction mechanisms are identical to framework implementations. For example, it can be difficult for a programmer to know if a framework class is meant for subclassing by a plugin. And someone looking at the plugin code may have a difficult time comprehending the overall architecture since the framework defines the architectural skeleton.

Expressing and checking constraints. Researchers have developed good specification languages for libraries, but they are still working to identify the constraints that frameworks impose on plugins. Languages exist to express some constraints, such as FCL [Hou et al., 2004] design fragments [Fairbanks, 2007], and FUSION [Jaspan and Aldrich, 2009], but none of these is yet a complete solution. Constraints must still be checked, perhaps using static or dynamic analysis of the plugin source code.

Bridging programmatic and declarative. Many frameworks require plugins to write both source code and declarative statements. These two forms must be consistent, for example, an entry in an XML file must correspond to an identically named class in Java source code. Type checkers generally work in one or the other form but not cross forms.

Programming language support. Ideally, given the prevalence of frameworks, direct support for framework interaction mechanisms would be high priority requirements future programming languages. Such language support might include ways to help with problems mentioned above, including localizing framework-plugin interactions, specifying framework constraints, and bridging programmatic and declarative parts of plugins.

Architecture languages. Despite frameworks providing an architectural skeleton for applications, frameworks are a poor fit for existing architecture description languages (ADLs). While most ADLs have "port" elements that represent a means to communicate between components, these ports are narrow and shallow to enforce encapsulation. Framework interactions with plugins, in contrast, are *wide* and *deep*: to accomplish simple tasks, a plugin may need to interact with many classes in the framework and may need to traverse deep into the framework's objects. Consequently, it is difficult to satisfactorily represent framework-plugin interactions using most ADLs.

7. CONCLUSIONS

Frameworks have been on the rise during the past decade, making inroads into many domains of software engineering. And yet, they have done so without correspondingly significant research interest. This paper attempts to change this situation and allow researchers to focus more on the unique challenges that frameworks entail. To this end, the paper investigates a variety of current object-oriented frameworks, provides a working defintion of what frameworks are and are not, categorizes frameworks along several dimensions, and identifies a number of interaction mechanisms, which are ways in which frameworks and their plugins interact. Interaction mechanisms can be implemented in a variety of different ways, from design patterns and strongly typed interfaces to reflection, annotations, and even URLs. We hope that our categorizations and interaction mechanisms can serve as a common vocabulary for dis-

cussing frameworks that is independent of programming languages and specific implementation details that other researchers will extend and build upon.

Acknowledgments

The authors recognize Bill Scherlis for his observation of deep-wide versus shallow-narrow interfaces, and thank <others>, and Morgan Stanfield for their comments on drafts of this paper.

References

K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP'09)*, pages 195–219. Springer-Verlag, July 2009.

R. Bunting. Bridging the framework modeling and implementation gap. *IBM Systems Journal*, 39(2):267–284, 2000. ISSN 0018-8670.

Facebook, 2009. URL `http://developers.facebook.com/`.

G. Fairbanks. *Design Fragments*. PhD thesis, Carnegie Mellon University, 2007.

G. Fairbanks. *Risk-Centric Software Architecture*. Taylor and Francis, 2010. URL `http://rhinoresearch.com/book`.

M. Fontoura, W. Pree, and B. Rumpe. UML-F: A modeling language for object-oriented frameworks. In *ECOOP '00: Proceedings of the 14th European Conference on Object-OrientedProgramming*, pages 63–82, London, UK, 2000. Springer-Verlag. ISBN 3-540-67660-0.

M. Fontoura, W. Pree, and B. Rumpe. *The UML Profile for Framework Architectures*. Addison-Wesley Professional, 2001. ISBN 0201675188.

M. Fowler. Inversion of control, June 2005. URL `http://martinfowler.com/bliki/InversionOfControl.html`.

E. Gamma and K. Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley Professional, 2003. ISBN 0321205758.

E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN 0201633612.

D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, pages 179–185, Seattle, Washington, April 1995.

A. Heydarnoori. *Supporting Framework Use via Automatically Extracted Concept-Implementation Templates*. PhD thesis, University of Waterloo, 2009.

D. Hou and H. J. Hoover. Towards specifying constraints for object-oriented frameworks. In *CASCON '01: Proceedings of the 2001 conference of the Centre for AdvancedStudies on Collaborative research*, page 5. IBM Press, 2001.

D. Hou, H. J. Hoover, and P. Rudnicki. Specifying framework constraints with fcl. In H. Lutfiyya, J. Singer, and D. A. Stewart, editors, *CASCON*, pages 96–110. IBM, 2004.

C. Jaspan and J. Aldrich. Checking framework interactions with relationships. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP'09)*, pages 27–51. Springer-Verlag, July 2009.

R. E. Johnson. Documenting frameworks using patterns. In *OOPSLA '92: Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, pages 63–76, New York, NY, USA, 1992. ISBN 0-201-53372-3. doi: http://doi.acm.org/10.1145/141936.141943.

R. E. Johnson. Components, frameworks, patterns. *SIGSOFT Software Engineering Notes*, 22(3):10–17, 1997a. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/258368.258378.

R. E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, 1997b. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/262793.262799.

R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1:22–35, June/July 1988.

OpenSpeedShop, 2009. URL `http://www.openspeedshop.org`.

D. Riehle. *Framework Design: A Role Modeling Approach*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 2000.