# Patterns for Sustaining Muddy Architectures

REBECCA WIRFS-BROCK, Wirfs-Brock Associates, Inc.
JOSEPH W. YODER, The Refactory, Inc.

Big Ball of Mud (BBoM) architectures are haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungles. They can be exceedingly complex and unwieldy to maintain. This paper presents two patterns for sustaining muddy architectures: PAVING OVER THE WAGON TRAIL and WIPING YOUR FEET AT THE DOOR. We will also recast PAVING OVER THE WAGON TRAIL into a third pattern that exacerbates the mud, PAVING OVER THE COWPATH, and explain why this is not an anti-pattern, but instead a darker form of well-intentioned tinkering.

Categories and Subject Descriptors: **D.2.2[Design Tools and Techniques]:** Object-oriented design methods;**D.2.11[Software Architectures]:**Patterns—*Big Ball of Mud*

General Terms: Design

Additional Key Words and Phrases: Big Ball of Mud, Sustainable architecture, DSLs

**ACM Reference Format:**

Wirfs-Brock, R., and Yoder, J. W., 2012. Patterns for Sustaining Muddy Architectures

## 1. INTRODUCTION

In 1998 the claim was made that the architecture that actually predominates in practice is: the BIG BALL OF MUD [Foote & Yoder]:

> "A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle. We've all seen them. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems."

Big Ball of Mud (BBoM) architectures are often viewed as the culmination of good design intentions gone wrong that result in a system that is hodgepodge of steaming, smelly anti-patterns. Yet how BBoM architectures come into existence and successfully evolve over time is much more nuanced. BBoMs often do not result from well-intentioned design ideas gone wrong. Nor are they simply an accretion of expedient implementation hacks. Often BBoM systems can be extremely complex, with unclear and unstable architectural boundaries and requirements.

Because of their complexity, BBoM architectures are likely not understood by any single mind. They typically are fashioned out of many parts, which together comprise a sprawling whole. So BBoM systems can and do have good, as well as bad and ugly parts.

In complex systems, software design decisions and architectural structures accumulate. Some good intentions do go awry. Some parts may have been brilliantly crafted and continue to be well designed, in spite of the mud surrounding them. Successfully deployed BBoM systems continue to work well enough, in spite of their design flaws. How can that be?

### 1.1 Contribution

In this paper we present two patterns for sustaining muddy architectures: PAVING OVER THE WAGON TRAIL and WIPING YOUR FEET AT THE DOOR. We will also discuss how PAVING OVER THE WAGON TRAIL can possibly exacerbate the mud by morphing into PAVING OVER THE COWPATH. This is a darker form of PAVING OVER THE WAGON TRAIL where well-intentioned tinkering went awry.

## 2. PATTERN: PAVING OVER THE WAGON TRAIL

### 2.1 Context

You observe a routine programming task being repeatedly performed. The code isn't difficult, but is tedious and time consuming to write. Copy-paste reuse doesn't lessen the time to write the code because many special cases need to be considered.

### 2.2 Problem

How can you make it easier to perform tedious, repetitive programming tasks for a system?

### 2.3 Forces

There are several forces:

- *Existing Codebase:* Large amounts of existing code include this tedious code.
- *Ongoing Development:* Programmers routinely add similar code to the system.
- *Limited Options for Refactoring:* Refactoring the existing code to eliminate repetitive code is discouraged and often impractical.
- *Limited Value to Refactoring:* It isn't obvious that refactoring the existing code will make the programming task any less tedious.
- *Defined scope:* The programming task is focused on specific well-known system functionality.
- *Duplicate code*: Just copying something and tweaking can be easy but maintaining code that is duplicated can become very cumbersome.

### 2.4 Solution

Provide a way to generate or describe the repetitive task. Sometimes this is through code generation or using some "descriptive" data that is interpreted. The following lists several potential solutions that can also be applied in combination to ease tedious programming tasks:

- Develop a domain-specific language that is focused on the specific programming task, or
- Develop a tool that generates code from a higher-level specification, or
- Develop a wizard tool that steps a developer through the process of specifying configuration data or code, or
- Develop a framework and/or runtime environment that that enables developers to simplify their programming tasks by providing higher-level support, or
- Identify and use existing tools or frameworks that provide such higher-level support.

GUI builders and persistent frameworks are common examples where these techniques are applied. You can define the GUI through a WYSYWIG editor and overwrite properties etc. Examples of this are Visual Studio's form designer and for Java, JFormDesigner. These tools generate code and hook methods where you can quickly build a GUI without all the low level programming to describe every detail. Some GUI builder tools provided a comprehensive platform for building the GUI, developing the supporting code, and integrating the GUI with event handling and other related programming tasks. SWING, GWT, and NetBeans are examples. While a GUI builder can save time in initially constructing the GUI, there can be drawbacks to their use. Sometimes using GUI builders makes maintenance more difficult, especially if the generated code, which can be hard to read, needs to be read and understood in order to debug it, or to make changes and enhancements.

Possibly ENTER IN AN ARCHITECTURAL DIAGRAM HERE….

### 2.5 Consequences

Advantages:

- Domain-specific languages or higher-level frameworks allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. This can reduce the amount of programming required to solve the programming tasks.
- Domain-specific languages allow validation at the domain level reducing trivial programming errors.
- Tools can perform consistency checks too, eliminating cut-copy-paste-then-modify errors.

– Code generators can make sure the code is generated properly to the framework and give a good working example to evolve from.

Disadvantages:

– Cost of learning a new tool or DSL or higher-level framework.
– Cost of designing, implementing, and maintaining a domain-specific language and tools.
– Finding, setting, and maintaining proper scope for the DSL or tools.
– Programmers are removed from low-level code and lose track of what is actually generated.
– Generated code can be harder to debug or performance tune.
– Adding behavior that has to be tightly integrated with or extends code that is generated or included with the DSL can become cumbersome.

## 2.6 Examples

WindowBuilder, as shown in Figure 1, is an Eclipse plug-in composed of SWT Designer and Swing Designer that makes it easy using WYSIWYG designer tools to create various forms and windows without writing a lot of code. The tool generates Java code. In addition to providing support for laying out and organizing the UI, the tool also supports UI actions by enabling event handlers to be attached to UI controls. WindowBuilder also supports changing various control properties using a property editor as well as internationalization.

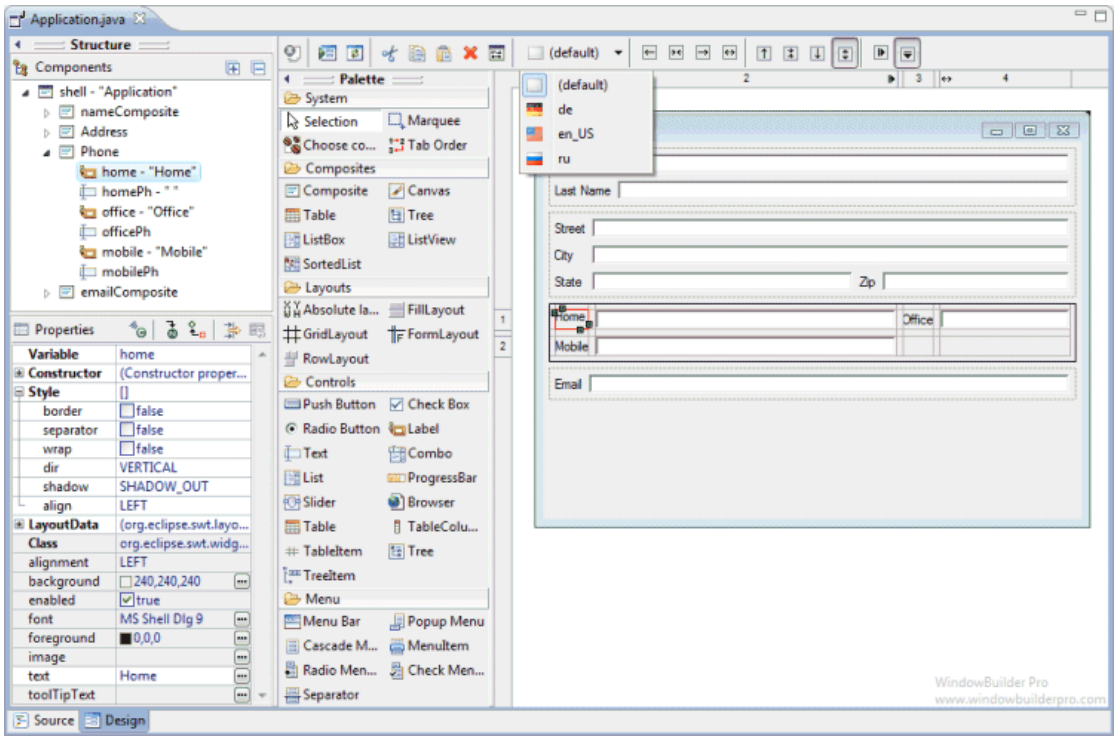


**Figure 1 - Window Builder**

Similarly, object-relational mapping tools like Hibernate or NHibernate provide powerful tools for quickly mapping objects to relational tables (see Figure 2 - NHibernate Designer). These systems help PAVE OVER THE WAGON TRAIL by eliminating much tedious programming. However, sometimes they do so at the expense of being able to comprehend how they actually implement data mapping and database queries, thus making it more difficult to tune database queries or optimize data cache performance.

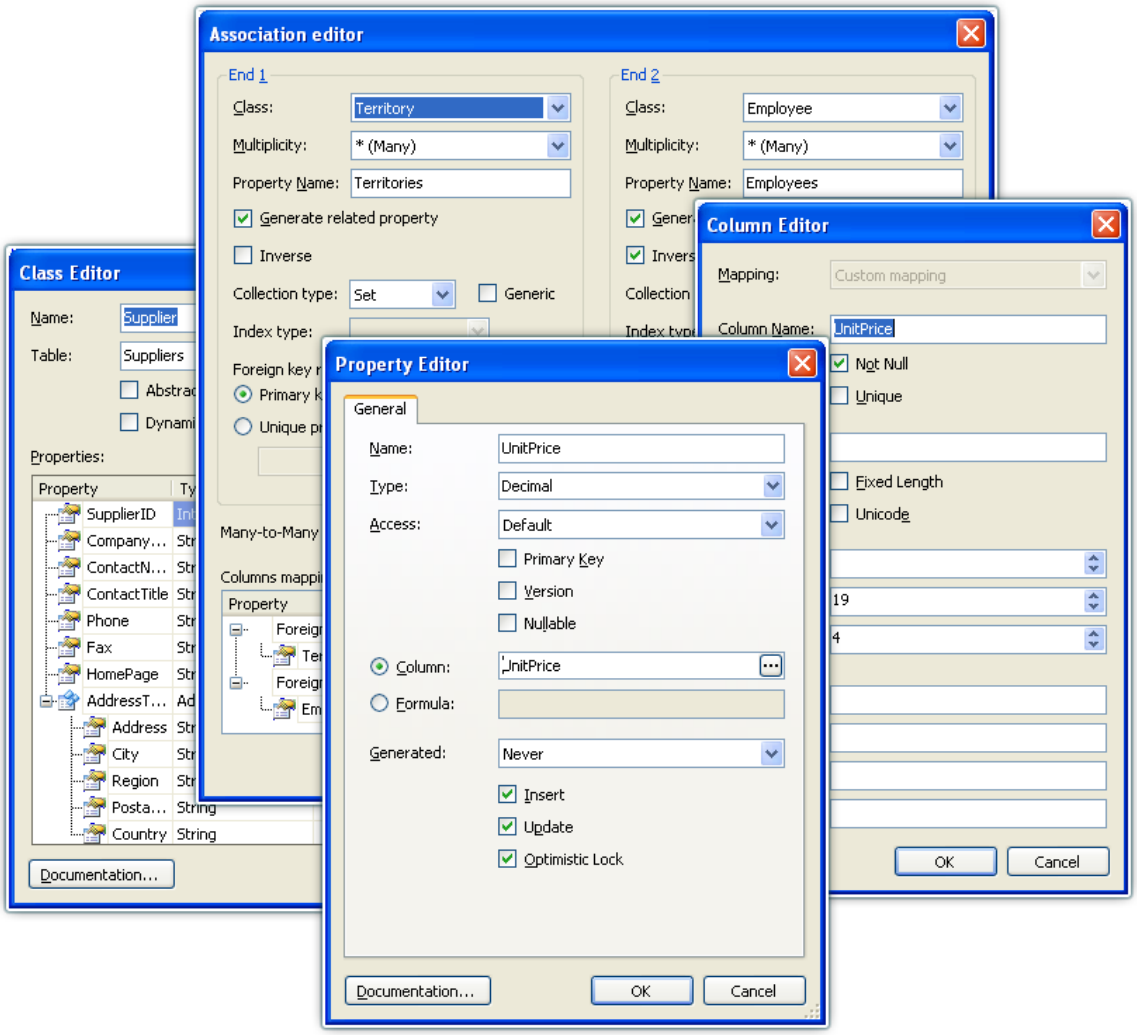


**Figure 2 - NHibernate Designer**

XML can be seen as another example of PAVING OVER THE WAGON TRAIL. XML evolved from HTML and Standardized Generalized Markup Language (SGML) created by Charles Goldfarb, along with Ed Mosher and Ray Lorie in the 1970s while working at IBM [Anderson, Collins]. SGML, despite its name, is not a markup language but rather a language for describing markup languages. HyperText Markup Language (HTML) invented by Tim Berners Lee was developed the late 1980s and became a popular application of SGML. However popular and well-used, HTML is, it is ill-suited for defining general purpose data storage types or describing new types even though many use it to do so. XML is an example of PAVING OVER THE WAGON TRAIL that was built upon HTML as a way to provide higher-level support for data storage and interchangeability. There are many XML development tools developed to support reading, writing, and validating XML (see Figure 3 - XML Development Tools). The problem with XML is that maintaining a large number of XML definitions can become unwieldy. Reading and editing XML data definitions can be very cumbersome and hard to maintain. Fortunately, many paving techniques such as editors and support tools have been developed to assist with this task.

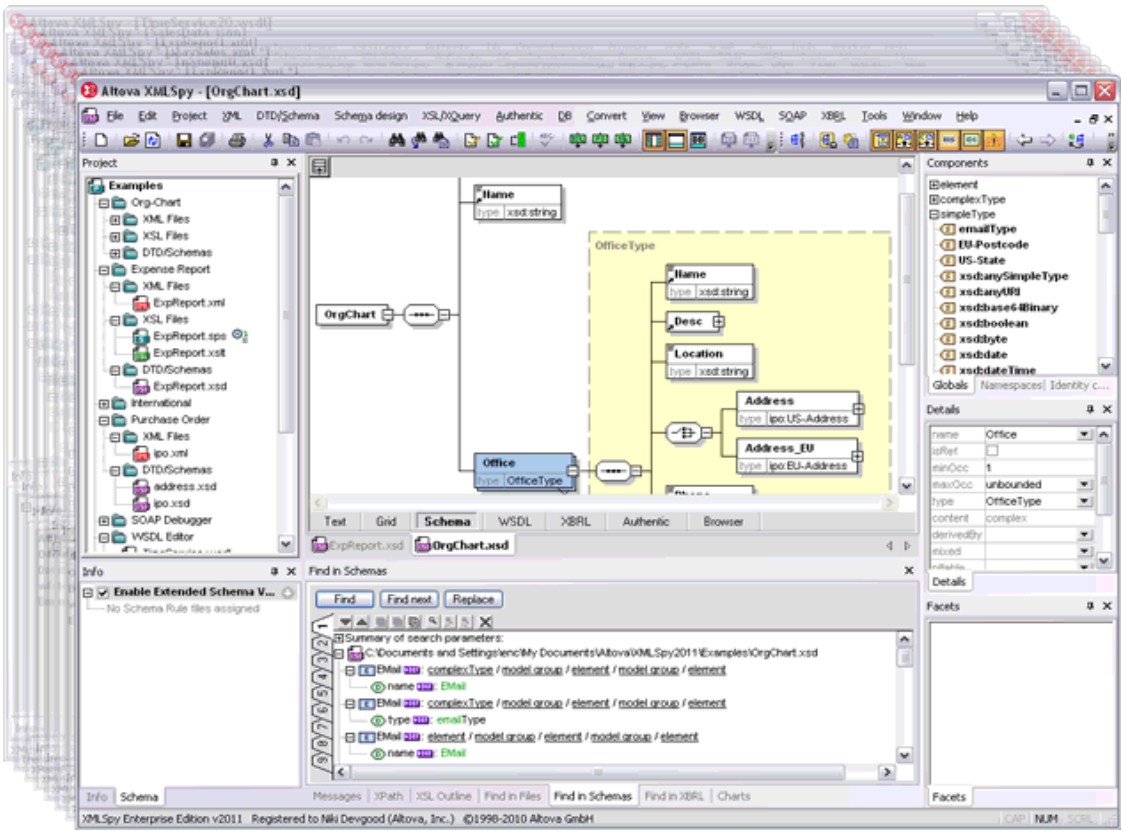


**Figure 3 - XML Development Tools**

Another example of PAVING OVER THE WAGON TRAIL is the inception and evolution of JSON. Douglas Crockford was the first to specify the JSON format [JSON]. JSON was an answer in the Javascript world to overly-complex XML. Javascript needed something like XML to define data structures to be transferred between the browser and other systems. Although JSON is based on a subset of the JavaScript scripting language (specifically, Standard ECMA-262 3rd Edition—December 1999[4]) commonly used by JavaScript programmers, it is a language-independent data format and has become widely used in other environments. Code for parsing and generating JSON data is readily available for a large variety of programming languages.

Crockford had an initial goal of making JSON efficient and less ambitious than XML and only support what was clearly needed. So the first implementation of JSON relied on the Javascript eval to parse JSON declarations. This limited the data types it supported.

As a consequence, JSON declarations represent numbers, booleans, strings, null, and arrays (ordered sequences of values) and objects (string-value mappings) composed of these values (or of other arrays and objects). But it does not support more complex data types such as function declarations, regular expressions, or Javascript dates. Date objects by default serialize as a string containing the date in ISO format, so while the full date formatting isn't preserved when using JSON, the basic date information isn't completely lost. If you need to preserve such values or support other types, you can programmatically transform values as they are serialized, or prior to deserialization, thus enabling JSON to represent additional data types.

In the spirit of minimal functionality, these data type limitations were carried forward in subsequent implementations that did not rely on built in eval functions to interpret JSON declarations. These limitations still exist today. Yet JSON is widely used because what it does is useful, although minimal. While PAVING OVER THE WAGON TRAIL with minimal functionality makes it easier to support different JSON implementations, there are drawbacks. One downside of JSON's minimal support for data type declarations is that there has to be an agreed upon "convention" between data definitions and deserialization/serialization implementations in order to provide support for other types that are not in JSON.

The Refactory, Inc developed a tool for one of their clients that allowed them to specify different invoicing rules. The Invoicing System was developed to be very adaptable and is based upon a reflective architecture [Yoder and Johnson] that allowed both power-users and end-users to change descriptive information about different client-specific invoicing rules. Originally it was cumbersome to change some of the rules through editing different values in a database and changing XML and XSLT. We Paved Over the Wagon Trail on this system by developing a visual language for defining the rules and for editing and validating the changes both from the database and from XML and XSLT. This paving allowed for very quick changes that could be validated by end users. There was also some well-defined hook points [Acherkan et al] where you could add new behavior by writing new C# code or XSLT. To implement a hook point required a deeper understanding of the architecture and was only needed when the existing rules or framework did not provide enough power for the client-specific invoicing needs.

### 2.7 VARIANT: PAVING OVER THE COWPATH

A problem can arise when the architect or developer over-abstracts or tries to do too much in their solution. Quite often developers have an itch to make tools or tooling in anticipation of saving programmers a lot of time. While the intent is to make life easier and minimize interactions with certain parts of the system, this can sometimes be overdone. As a consequence what was once simple, but tedious before, becomes even harder. We call this PAVING OVER THE COWPATH. There might be temptation to pave over the cowpath and provide a really cool tool. If you are alert, you quickly learn that the cows do not want to travel on the path you've newly paved. Persisting with a tool-based solution when all signs point to it not being useful can be seen as an anti-pattern where an attempt at PAVING OVER THE WAGON TRAIL goes awry.

Your main objective was to find a way make simple tasks easier. There might be several potential solutions (and they can also be combined). For example you might try to develop an adapter or bridge that takes the "muddy requests" and translates them into calls that preserve the "clean interface". You want the ability to add customized muddy interfaces without breaking/changing existing code. The problem is that trying to use the new tools can be harder to use than the original programming approach and can lead to even more muddy code or complexities. It is important to analyze the tradeoffs before your start paving and to observe the effects of your paving solution on those who apply it. Will the benefits really outweigh the costs? Will the final solution make the code easier to maintain?

One example of Paving Over the Cowpath is the development of visual programming languages to construct database applications. Digitalk PARTS and VisualAge for PARTS were examples of two visual programming languages for Smalltalk. Users would construct an application by selecting components from a catalog and wiring them together to specify the flow of events and simple programming logic. Figure 4 shows a small example of this approach. While the idea of visually constructing simple database access applications seems productive, in practice many users did not have simple data access needs. To implement their applications, they would have to wire together many components, ending up with applications with hundreds of components and many more connections between them. Comprehending and maintaining large visual programs proved extremely difficult. The idea of visual programming in practice works only on small applications where connections between components are straightforward and the programming logic is simple. These visual programming tools did not have the intended productivity boost for anything but very simple applications.

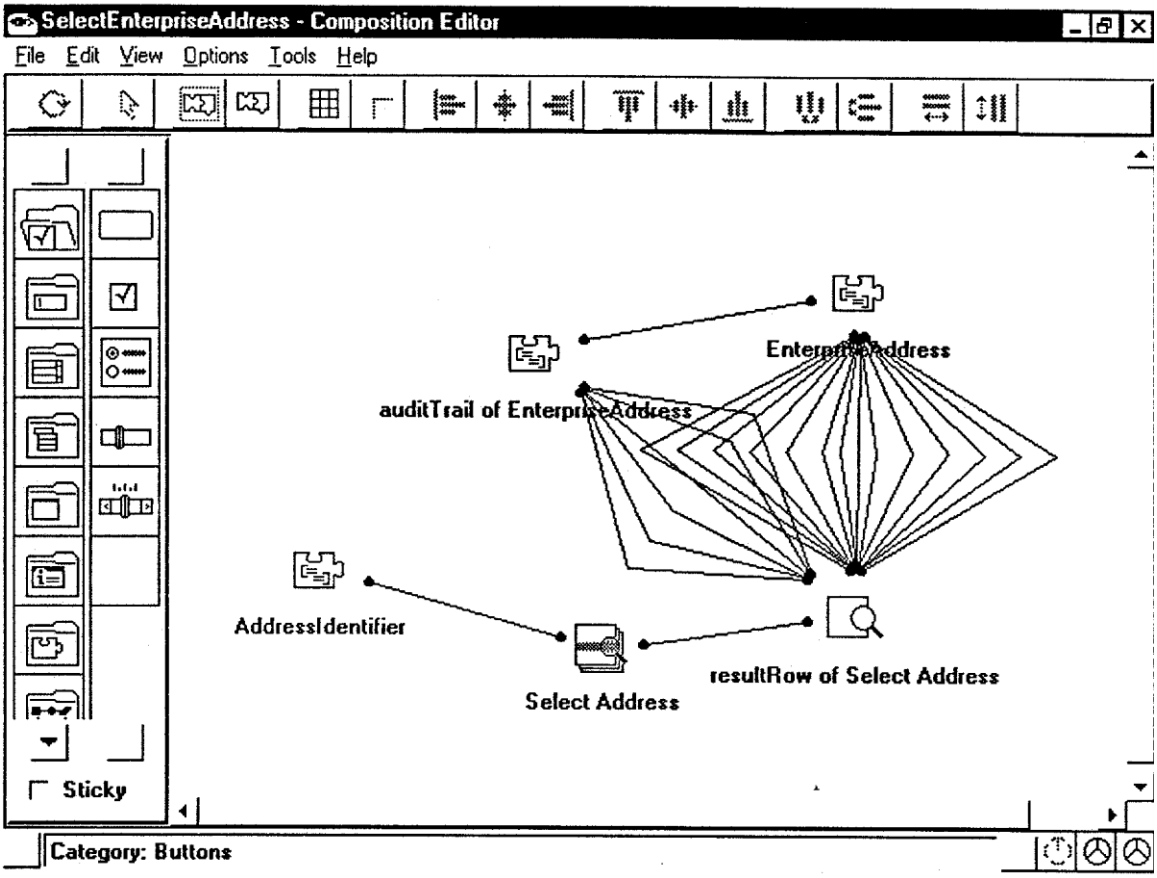


Figure 4 –A simple visual program example

### 2.8 Related Patterns

SWEEPING UNDER THE RUG protects outside callers from mud that is inside the system, while Paving Over The Wagon Trail is intended to hide complexity from programmer.

### 2.9 Known Uses

GUI & Persistent Frameworks such as WindowBuilder, JFormBuilder, NHibernate, and Hibernate.
External DSLs such as regular expressions, SQL, CSS, Postscript or LaTex.
Internal DSLs such as jQuery in Javascript or Rake in CoffeeScript.
Custom-built GUI for describing business rules saved as XML (State and for Refactory client)

## 3. PATTERN: WIPING YOUR FEET AT THE DOOR

### 3.1 Context

Some parts of your system are good, but there are pressures to change the API and functionality of existing good components in order to support other poorly designed components.

### 3.2 Problem

How can you avoid compromising the interface and design of a component and protect it from getting polluted by or coupled to other related components that interact with it?

### 3.3 Forces

There are several forces:

– *Pressure to develop interfaces required by muddy code*

– *Ongoing Development:* Programmers routinely add code to the system.
– *New functionality is being added*
– *Limited ability to change existing (possible muddy) components (outside your control)*
– *Need to accept requests from external systems*
– *Need to integrate existing (unchangeable) systems with your components*

### 3.4 Solution

Provide a specialized interface to your "clean" component, which performs data verification and possible filtering or data cleansing before delegating the call to the preserved "clean" interface of the component you don't want to be compromised. This can be done either through an adapter or bridge code that takes the (possibly muddy) requests and translates them into calls that preserve the "clean" interface.

Figure 5 shows one possible implementation of wiping your feet at the door where a separate component performs some cleansing, transforming, and/or data validation before delegating the request to the "clean" components. This pre-processing allows us to wipe the feet so that the core component's processing can be consistent and remain uncompromised. Another option would be to provide an additional "mud removing" interface to your clean component that filters and validates before delegating clean requests to the "clean interface". By providing a separate component, you will be able to deal with different variances of the client code including possibly having a security proxy for validating and protecting the internal component. Possibly you even have pluggable adapters with different rules depending upon the client code.

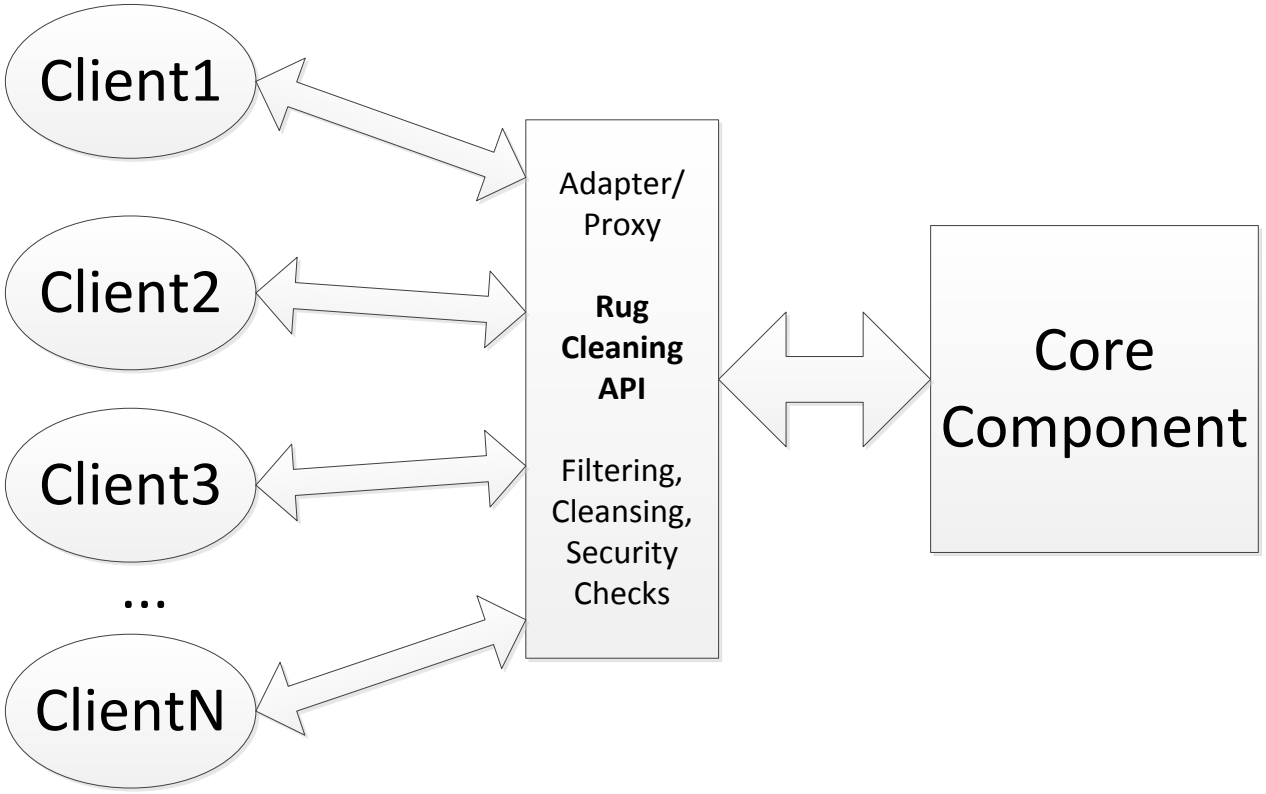


**Figure 5 –Generic Filtering Architecture View**

Without this new API cleansing the data, it would require tightly coupling the external systems with the internal component. This could muddy up the code and also make it much harder to add variances. For example you might envision where your internal component will have different variances depending upon different business requirements from varying clients. By separating this out we can keep those variances into a separate set of components (the adapters or proxies), thus allowing the common internal component to remain clean and thus focus on its main responsibility.

### 3.5 Consequences

Advantages:

- Preserves the integrity of the clean component and its desired interface
- Separates transformation/validation code from code that implements component functionality
- Ability to add one or more customized interfaces that support muddy code without breaking/changing existing code
- Ability to validate muddy requests without adding complexity to existing components.

Disadvantages:

- It may not be possible to completely contain the mud (especially when new behaviors need to be added to the component in order to support its muddy clients)
- Decreased performance with translation layers between components
- Increased complexity of an extra adaptation layer (such as a mediator) to interact with components (mud can start to grow there)
- Multiple interfaces (clean and muddy) increases system complexity

### 3.6 Examples

Figures 6 presents an architectural view for one of The Refactory's client's processing of importing inventory data and processing it according to client-specific business rules. For example, some clients could import customer-specific inventory data which came in various formats including flat files from main frames, CSV files, XM files, Excel files, etc. Each file format varies based on different client systems and their specific formatting rules. The core system for importing and processing such placing orders contained common code used regardless of file format being processed.

In the example below, if we are getting various formats of different customer files from different sites, we would first need to do some cleansing, transforming, and validating according to client specific rules. This pre-processing before we place the order is where we wipe the feet so that the core PlaceOrder process can be consistent. Any new client requirements can be done through this wiping (transformation) at the front door. Notice that the Place Order is the generic system component that is common, regardless of client.

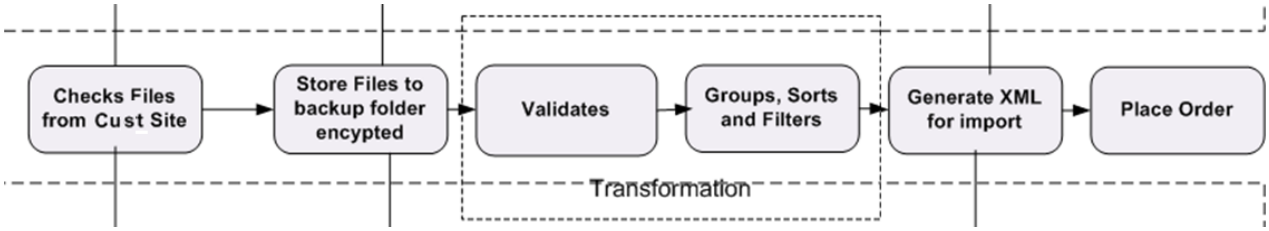


**Figure 6–Filtering and Cleansing Sequence to keep Place Order Interface Clean**

To prevent us from "muddying" up the core, we wrapped these systems and put in DYNAMICHOOKS [] where we could apply client-specific rules without changing the systems. Common processing of "standard format" in conjunction with transforming the imported data to a common formal via hook code was an effective way to adapt to constantly changing formatting requirements. The data was cleansed, validated, and modified before processing with similar internal code. This cleaning, transformation, and validation process is where we were wiping our clients' feet at the front door and preventing the client-specific code from compromising the integrity of our core processing code.

This was generalized to enable chaining multiple transformations that we applied according to client-specific rules to several different processes (see Fig 7).

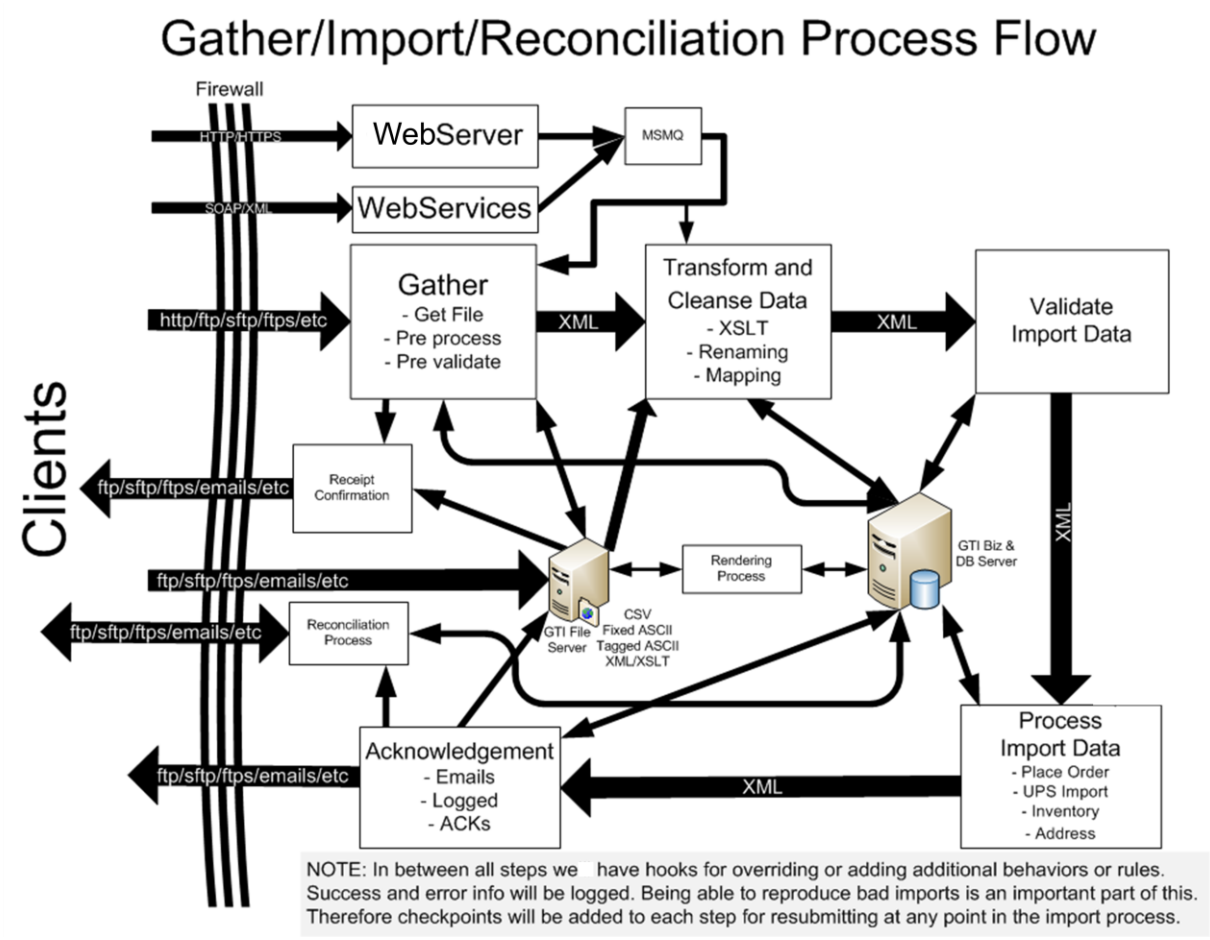


**Figure 7 –Generic Filtering and Cleansing Sequence for Various Processing**

Wirfs-Brock Associates worked with a telco client to implement a system, shown in Figure 8. This system integrated back office OSS applications such as telephony product ordering, provisioning and billing systems. The central component of the architecture, the Application Integration Services component, was responsible for processing orders. It would set up and monitor the status of tasks initiated in external systems and control the workflow for processing an order. Processing an order involved at a minimum provisioning or de-provisioning telephone equipment and adjusting billing. Each external system was integrated via an adapter, which took "raw" inputs from that system and converted them into canonical representations of orders, provisioning, and billing information requests. For any system that needed to be controlled, canonical requests were converted into specific external system APIs. Communications between any external systems were assumed to be untrusted, which meant that all requests and data had to be verified before being passed into the Application Integration Services component. Identifying trust regions [Wirfs-Brock] and trust relationships between external systems allowed us to simplify the architecture. Collaborations between the Application Integration Services and any adapter were designed to be trusted. One consequence of following the pattern of wiping your feet at the door using adapters meant that the implementation of the Application Integration Services code was not concerned about correct inputs, allowing it to be focused on task construction and monitoring.

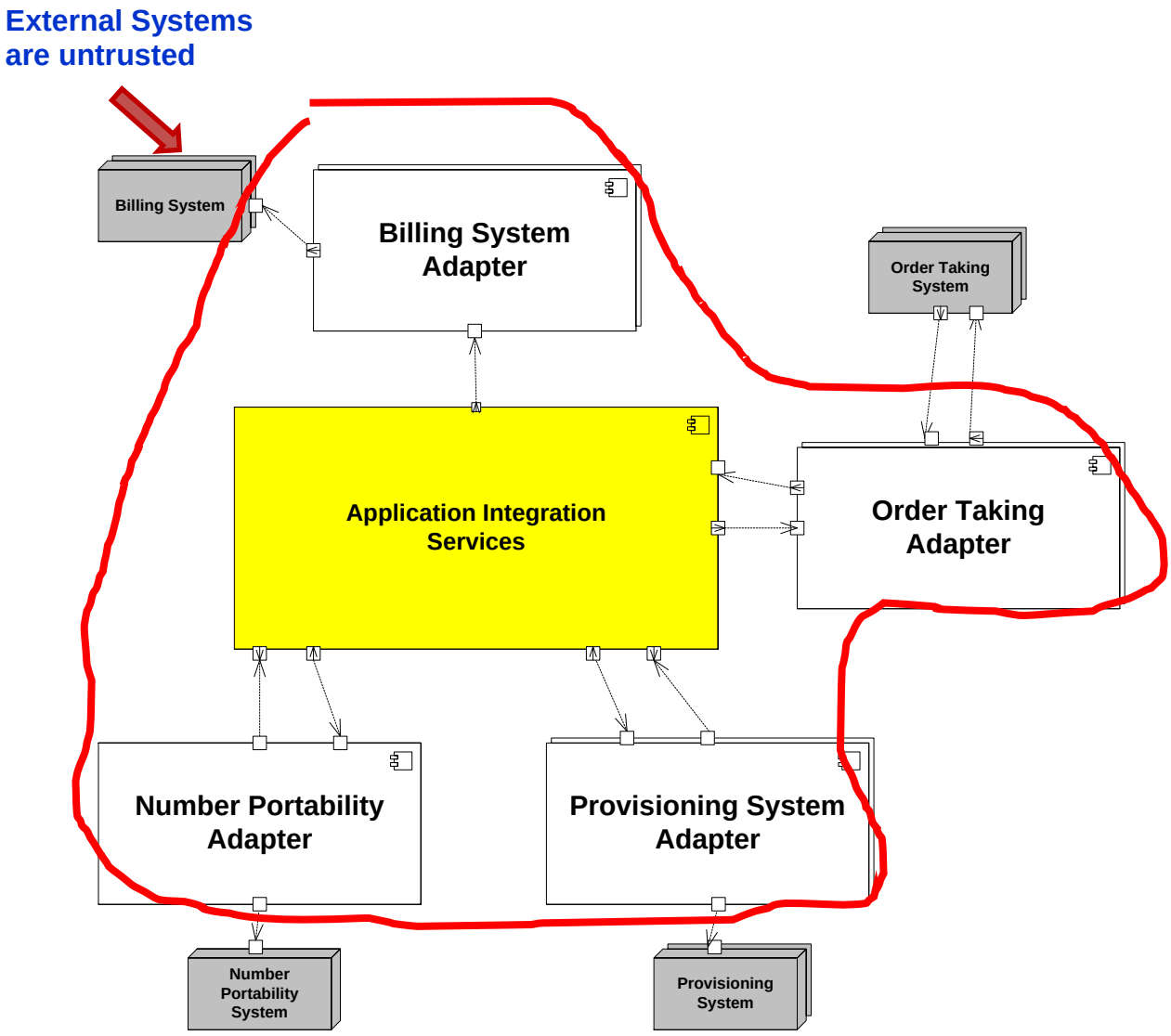


Figure 8 – Architecture of the OSS Integration System showing Trust Boundaries

### 3.7 Related Patterns

An ANTICORRUPTION LAYER [Evans] links two BOUNDED CONTEXTS- code you are writing and code in created by someone else that you have incomplete understanding of and little control over. An ADAPTER [Gamma et al] translates one interface for a class into a compatible interface.

ADAPTOR, BRIDGE, DECORATOR, PROXY [Gamma et al] and FILTER [Bushmann et al] are possible ways to implement WIPING YOUR FEET AT THE DOOR.

SWEEPING UNDER THE RUG [Foote and Yoder] is another way to hide the mess under the hood to protect outside systems.

### 3.8 Known Uses

Order processing systems by Refactory and Wirfs-Brock Associates mentioned in the above examples.

The Struts2 framework [Struts2] supports Java web application development. A Struts action is an instance of a subclass of an Action class, which implements a portion of a Web application. Most of the framework's core functionality is implemented via Interceptors. Features like double-submit guards, type conversion, object population, validation, file upload, page preparation, and more, are all implemented using Interceptors. Each

and every Interceptor is pluggable, so programmers can easily add their own interceptor objects to filter and process requests before they are received by actions.

The Eiffel programming language [ECMA] defines the notion of contracts. Bertrand Meyer introduced Design by Contract™ [Meyer] as well, in connection with the Eiffel programming language. This style of programming separates code that asserts pre and post-conditions and invariants from code that performs a specific function. This separation of concerns is in the spirit of wiping your feet at the door. One consequence of programming in this style is that code within a function can be written assuming data parameters conform to all conditions asserted by pre-condition contracts.

## 4. CONCLUSIONS

Much has happened in our industry since the original discussion and publication of the Big Ball of Mud. When that paper was originally published, agile methodologies such as eXtreme Programming [Beck] and Scrum [Schwaber and Beedle], were just beginning to gain popularity. We also have seen a growth of systems being built more with services, specifically through cloud computing. It was in 2001 that the Agile Manifesto [Manifesto] was published and agile practices have become common in our Industry. The growth and popularity of agile practices could have happened partially due to the fact that the state of the art of software architectures is so muddy. Being agile, with its focus on extensive testing and frequent integration, arguably makes it easier to deal with evolving architectures and keeping systems working while making significant improvements and adding functionality.

A common misconception of BBoMs that we want to dispel is that they are anti-patterns. This is not and never was the intent of the original authors. There are many forces and good reasons that can lead to a BBoM. In fact, architects and top development teams are often doing exactly the right thing when they end up with some mud in their systems. Perfection can be the enemy of "good enough" and it is often the case that something not as good wins [Gabriel]. An important point of the original paper is that often there are good decisions that can lead to muddy architecture—and only in hindsight can you see what might have been a better, less muddy solution. We should accept this aspect of building complex system rather than shun this, see why it happens, and focus more on how to deal with complex systems and how to make our code more habitable. This paper is our attempt at capturing proven practices for sustaining complex (often muddy) systems. We also feel these patterns are often appropriate for simplifying complex systems, whether your goal is to clean up or contain mud or not.

The original Big Ball of Mud paper described some patterns such as SHEARING LAYERS and SWEEPING IT UNDER THE RUG as a way to help deal with muddy architectures. This paper presents two other patterns PAVING OVER THE WAGON TRAIL and WIPING YOUR FEET AT THE DOOR as a means to sustain architectures, whether muddy or not, as well as continue to make code habitable.

## 5. ACKNOWLEDGEMENTS

We are grateful to Michael Stal for his valuable insight and input during the PLoP shepherding process.

## REFERENCES


Eli Acherkan, Atzmon Hen-Tov, Lior Schachter, David H. Lorenz, Rebecca Wirfs-Brock, and Joseph W. Yoder, "Dynamic Hook Points," *2nd Annual Asian PLoP Conference, Tokyo, Japan. October 2011*

Tim, Anderson. *Introducing XML,* http://www.itwriting.com/xmlintro.php

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns.* John Wiley, 1996.

Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison–Wesley, October 5, 1999.

Chris Collins, *A Brief History of XML,* http://ccollins.wordpress.com/2008/03/03/a-brief-history-of-xml.

ECMA Standard-367: Eiffel: Analysis, Design and Programming Language, 2nd Edition. June 2006.

Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003.

Brian Foote and Joseph Yoder, *Big Ball of Mud*, Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97) Monticello, Illinois, September 1997.

Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Professional, Nov. 1994

Richard Gabriel, *Worse is Better,* http://www.dreamsongs.com/WorseIsBetter.html

Richard Gabriel, *Lisp: Good News, Bad News, How to Win Big,* http://dreamsongs.com/WIB.html

Hibernate, http://www.hibernate.org.

JFormBuilder, http://sourceforge.net/projects/j-form-builder.

JSON, http://www.json.org.

Manifesto for Agile Software Development, http://agilemanifesto.org

Bertrand Meyer, *Touch of Class: Learning to Program Well with Object and Contracts,* Springer-Verlag, 2009.

NHibernate, http://nhforge.org/Default.aspx.

Ken Schwaber and Mike Beedle, *Agile Software Development with Scrum.* Prentice-Hall, 2001.

Struts2 http://struts.apache.org/2.x/docs/home.html

Rebecca Wirfs-Brock and Alan McKean, *Object Design: Roles, Responsibilities, and Collaborations.* Addison-Wesley, 2002.

WindowBuilder, http://www.eclipse.org/windowbuilder.

Joseph Yoder and Ralph Johnson, "The Adaptive Object Model Architectural Style," Proceedings of The Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02).