Patterns for Sustaining Architectures

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

Unless ongoing attention is paid to architecture, as complex systems evolve to meet new requirements they can become unwieldy to maintain and devolve into poorly architected Big Balls of Mud. This paper presents two patterns for sustaining software architectures in the face of increasing complexity: 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. These patterns help with both mud prevention and sustaining complex, evolving architectures.

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

General Terms: Architecture, Design, Patterns

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 Architectures. 19th Conference on Pattern Languages of Programs (PLoP), Tucson, Arizona, USA (October 2012), 13 pages.

1. INTRODUCTION

Software design decisions and architectural structures accumulate. Some parts of a complex software system may have been brilliantly crafted and continue to hold up well after many modifications. However, in most complex system some parts don't hold up so well. Over time, even a great design can be compromised by successive architectural revisions. 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 many design decisions that, over time, result in a system that is hodgepodge of steaming, smelly anti-patterns. Yet how BBoM architectures come into existence and successfully evolve 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 have good, as well as bad and ugly parts. Successfully deployed BBoM systems continue to work well enough, in spite of their design flaws.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers' workshop at the 19th Conference on Pattern Languages of Programs (PLoP). PLoP'12, October 19-21, Tucson, Arizona, USA. Copyright 2012 is held by the author(s). ACM 978-1-4503-2786-2

When trying to sustain architecture, it is important to try to protect certain parts of the design. Parts of large systems will change at different rates. There are certain actions you can take to isolate and put boundaries around different parts of your system, thus making it easier to sustain a good architecture while making significant changes.

In this paper we present two patterns for sustaining architectures: PAVING OVER THE WAGON TRAIL and WIPING YOUR FEET AT THE DOOR. These patterns, when employed help you sustain architectures in general. They can help with preventing mud from creeping into your design and they can also help you deal with improving evolving systems that might already have muddy parts. We will also discuss how PAVING OVER THE WAGON TRAIL can possibly exacerbate the muddiness of an architecture by morphing into PAVING OVER THE COWPATH. This is a darker form of PAVING OVER THE WAGON TRAIL where well-intentioned tinkering goes awry.

2. PATTERN: PAVING OVER THE WAGON TRAIL

ALIAS: MAKE REPETITIVE TASKS EASIER / STREAMLINING REPETITIVE CODE TASKS

2.1 Context

You observe a routine programming task being repeatedly performed. The code might not be 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. Often the code you are writing is very similar with minor variances. Writing this code can be monotonous yet take time and attention to get correct.

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:* As you evolve any system you need to consider what to do with existing code that still works but that could be improved. Do you leave it alone or refactor it to fit better into your new architecture?
- *Ongoing Development:* Programmers may routinely add similar code in support of new features. How can you safely rework and evolve the architecture with minimal impact?
- *Limited Options for Refactoring:* Refactoring to eliminate repetitive code is may be discouraged. Why refactor something that is currently released and working? Should you evolve parts of the system and still leave existing working code alone?
- *Limited Value to Refactoring:* It isn't obvious that refactoring some existing code will make the programming task any less tedious or error prone. How do you know that the new way of implementing the functionality will actually make development easier and faster?
- *Defined scope:* Sometimes a tedious programming task is focused on specific well-known system functionality. This makes it feasible to consider adding higher-level support for that functionality. How might you streamline the process of adding new code in this case?
- *Duplicate code*: Copying some code and tweaking it can be straightforward. However, maintaining a large codebase that has been replicated and then modified can become cumbersome. What are the tradeoffs and how can you get the best win for your investment in restructuring the code? If the repetitive code isn't really changing much, maybe it isn't worth worrying about improving how it works.

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 can be interpreted. Sometimes a combination of these techniques can be used. Following are a list of several potential solutions that can be applied to ease tedious programming tasks, ordered in terms of increasing difficulty:

- Create simple examples, templates, and scripts to show developers how to write the code, or
- Identify and use existing tools or frameworks that provide higher-level support instead of writing tedious, low-level code, or
- Develop a tool that generates code from a higher-level specification, or a wizard tool that steps a developer through specifying configuration data and code, or

- Develop a framework and/or runtime environment that that enables developers to simplify their programming tasks by providing higher-level support, or
- Develop a domain-specific language that is focused on the specific programming task.

There are many things to consider when deciding on an appropriate solution. Our advice is to do the simplest thing possible that minimizes your maintenance effort. This includes both the effort required to maintain your tools and your existing codebase. For example, you may develop a tool to generate code from a higher-level specification and at the same time decide not to convert any existing code to the "new" way. You may decide this is reasonable because the existing code already works, it may not be changing much, and you don't want to take the time to convert it.

A solution that employs a new framework or new DSL can require significant design and reimplementation effort, but also opens up the possibility of improving the architecture to handle the functionality more efficiently. This raises the question of when you should convert existing tedious, but working code, to use a new framework or DSL and when should you leave pre-existing code alone, resulting in a potentially muddier architecture, which now supports multiple ways of accomplishing the same task.

Any solution that enables you to leave working code untouched opens up the potential for muddying the architecture with support for both new and old ways of performing a programming task. But converting large amounts of existing code to the new way can take time and be error prone. Even if the old way is declared obsolete and no longer used, you've still muddied up your architecture with multiple ways of doing things.

Supporting two approaches to the same programming task increases your maintenance effort, simply because you have to know both in order to maintain your system. Even more daunting, if you continue this strategy of leaving stable code alone and only implementing new functionality using the new approach, you may end up years later with multiple approaches supported by your architecture to accomplish the same task. It is often prudent in the short term to not convert old code to use new tools and techniques; it may not be wise in the longer term.

GUI builders and persistence 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 requiring low level programming to describe every detail. Some GUI builder tools provide 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.

2.5 Consequences

Independent of the tool, framework, or DSL-based solution you choose there are positive consequences:

- Repetitive tasks that used to be tedious have been simplified.
- The overall amount of cut-copy-paste-then-modify reuse of code has been reduced.
- Refactoring any new code that uses the new solution can be easier since it is utilizes a framework or improved architecture.

And independent of these solutions, there are some common disadvantages:

- Finding, setting, and maintaining proper scope for your solution can be difficult.
- Converting existing code to the new solution can be time consuming.
- Enhancements to your new tools, frameworks or DSL might force you to have to regenerate working code or evolve working code.

Additionally, there are some consequences that are specific to several approaches.

Advantages:

- Simple examples and templates can be easy to use. When well done you know exactly what to do: just fill in the blanks, modify specific lines in a configuration file, and specify these parameters.

– Tools offer a common place to evolve what is generated so that maintenance can be simpler. Tools and scripts can perform consistency checks, eliminating cut-copy-paste-then-modify errors. Tools and wizards can also make sure that programmers complete all the steps of a tedious programming task.
– Code generators can ensure that the generated code properly uses existing frameworks and provide a good working example of how a task should be performed.
– Domain-specific languages and 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 effort required to solve the programming tasks. Domain-specific languages also allow validation at the domain level, thus reducing trivial programming errors.

Disadvantages:

– Code examples and templates lead to cut-copy-paste reuse and don't reduce the amount of repetitive code.
– Generated code can be harder to debug or performance tune. Adding behavior that has to be tightly integrated with or extends such code can be difficult. Programmers can become too removed from low-level code and lose track of what is actually generated.
– There can be a steep learning curve for a new tool or DSL or higher-level framework.
– The cost of designing, implementing, and maintaining a domain-specific language or tool can be high.
– Frameworks and DSLs can have performance limitations.

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 internationalization and changing various control properties using a property editor.

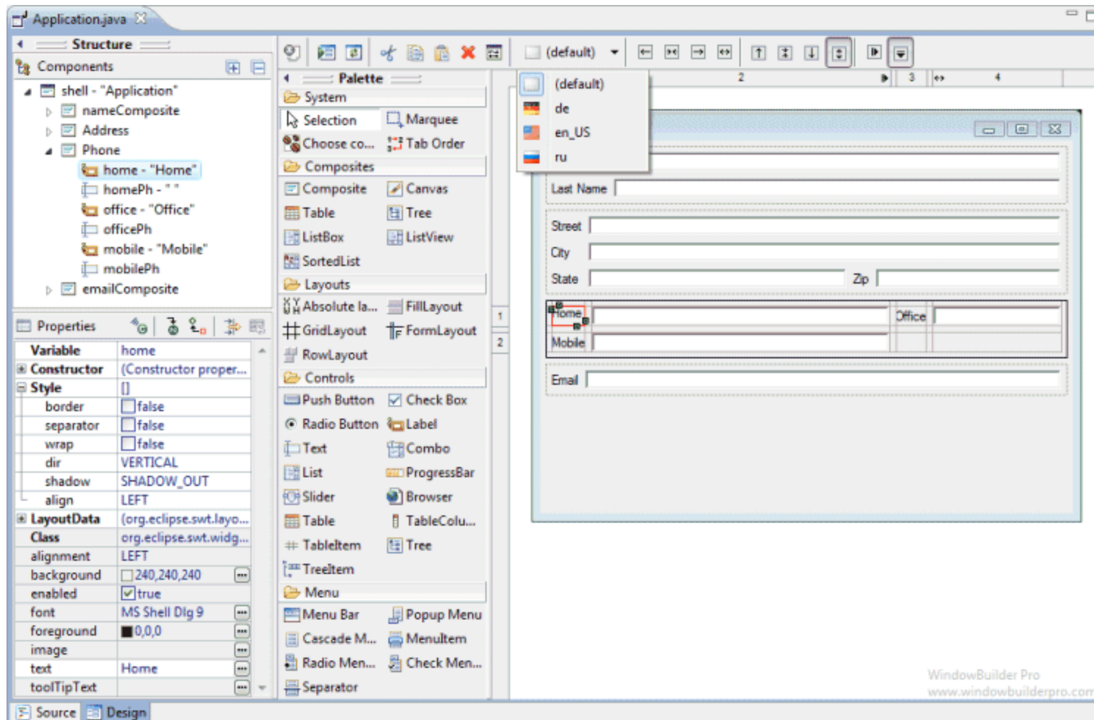

Figure 1 - Window Builder

Similarly, object-relational mapping tools like Hibernate or NHibernate offer powerful tools for mapping objects to relational tables (see Figure 2 - NHibernate Designer). These systems help Pave Over the Wagon Trail by eliminating much tedious programming. However, they do so at the expense of being able to understand the details of how data mapping and database queries are implemented, thus making it more difficult to tune database queries or optimize data cache performance.

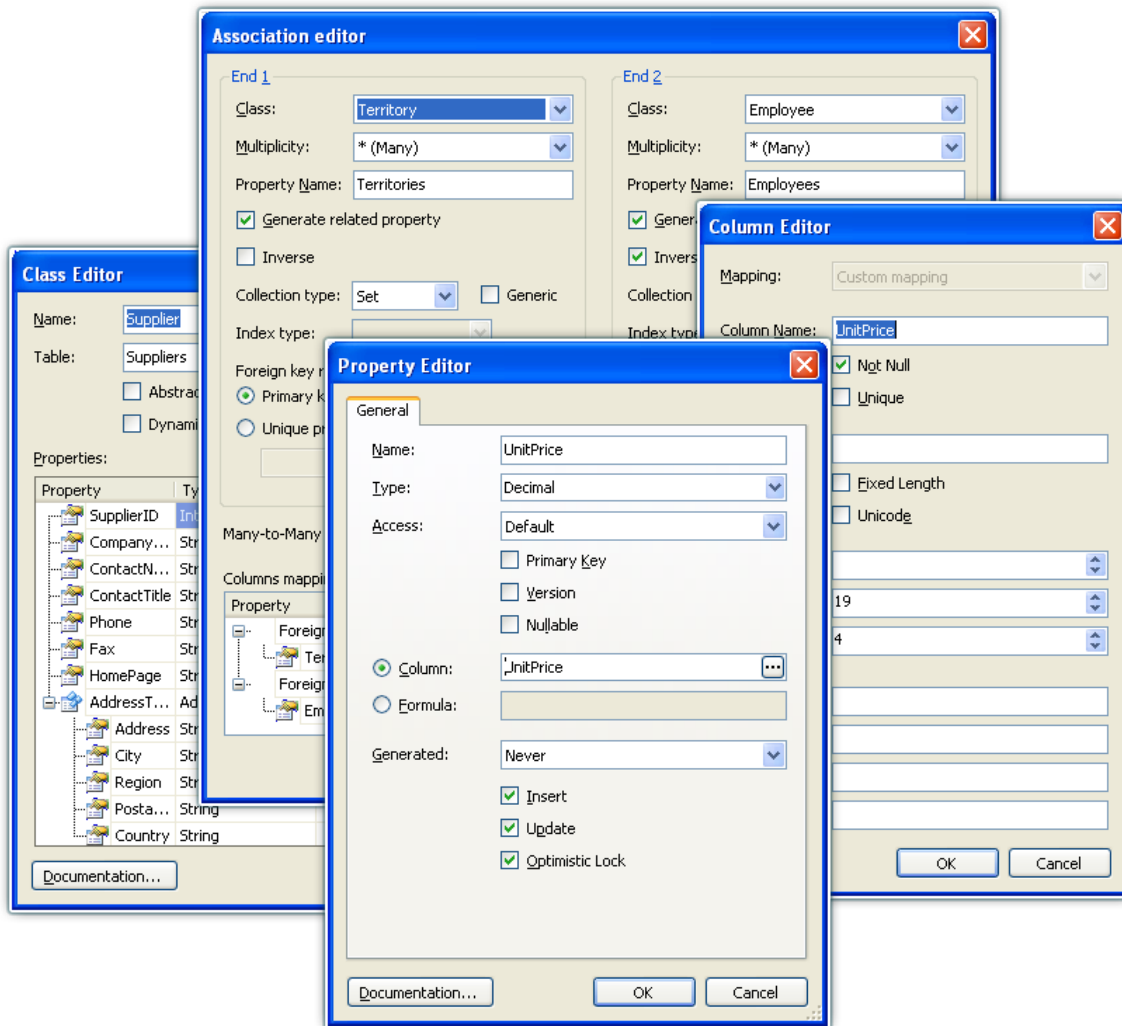

Figure 2 - NHibernate Designer

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]. HyperText Markup Language (HTML), invented by Tim Berners Lee in the late 1980s, became a popular application of SGML. XML is an example of Paving Over The Wagon Trail that built upon HTML 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).

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. Maintaining a large number of lengthy XML definitions can become unwieldy. And reading large XML data definitions is tedious. Fortunately, many paving techniques such as editors and support tools have been developed to assist with maintaining and editing XML definitions. Tool-based solutions can make it easier to do simple routine tasks, but they don't make everything easier. For example, extensive changes to data definitions or debugging complex XML specifications are still difficult.

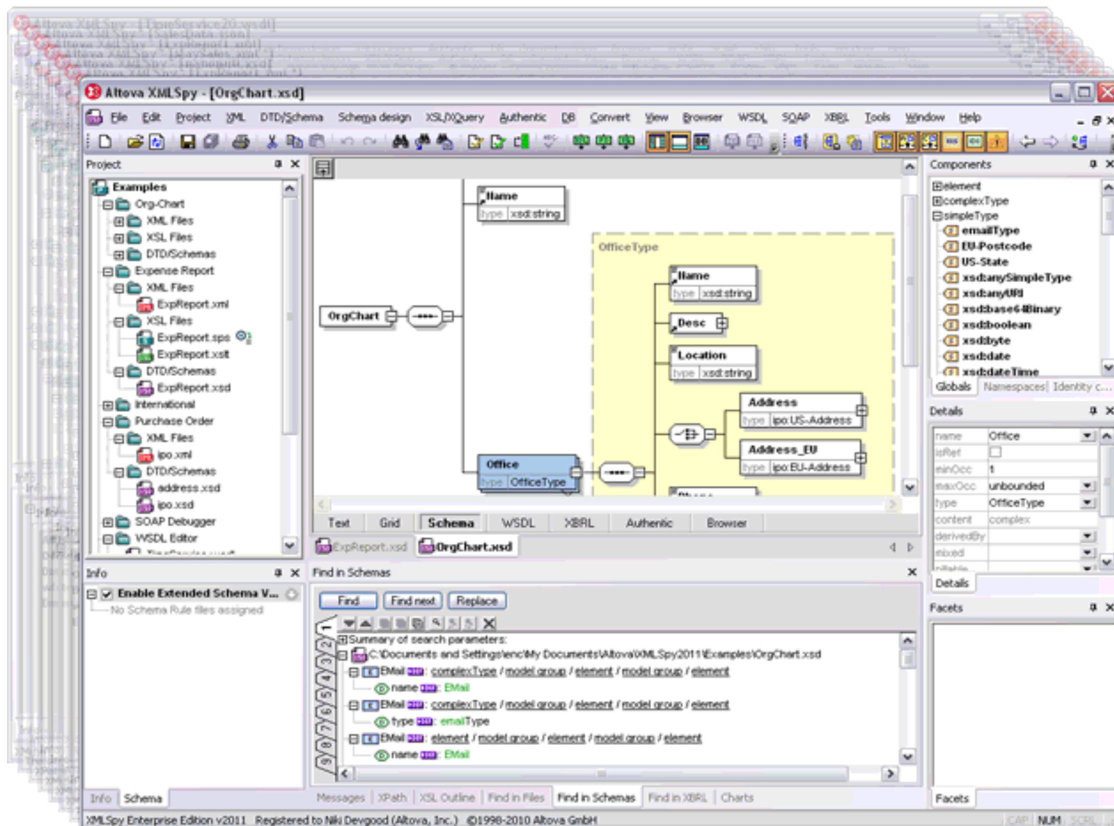

Figure 3 - XML Development Tools

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 originally 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 that 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. Crockford wanted to only support what was clearly needed. The first implementation of JSON relied on the JavaScript eval to parse JSON declarations and 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 the JavaScript eval function to interpret JSON declarations. These limitations still exist today. Yet in spite of these limitations, JSON is widely used because what it does is useful, although minimal. JSON paved over the same wagon trail as XML did, but with different objectives. JSON is useful, but limited in its scope. While PAVING OVER THE WAGON TRAIL with minimal functionality makes it easier to support different JSON implementations, there are a few 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 part of JSON.

A third example of PAVING OVER THE WAGON TRAIL is a framework developed by The Refactory, Inc. that allowed a client to specify different invoicing rules. The Invoicing System was developed to be very adaptable and was 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 by testing the results of invoicing clients based upon the new rules. Occasionally there were requirements that were beyond the capability of the core framework. To handle these requirements part of the framework used some well-defined hook points [Acherkan et al] where new behavior could be added by writing new C# code or XSLT. Implementing 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

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. When the architect or developer over-abstracts or tries to do too much in their solution, sometimes what was once simple but tedious 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 that 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. However, trying to use the new tools might be harder to use than the original programming approach or 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. In practice, visual programming works well only for small applications where connections between components are straightforward and the programming logic is simple. These visual programming tools only boosted productivity for very simple applications.

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 programmers, preventing mud from creeping inside the component.

This pattern is achieving similar results as the VISUAL BUILDER and LANGUAGE TOOLS patterns as outlined in Evolving Frameworks, A Pattern Language for Developing Object-Oriented Frameworks [Roberts and Johnson].

DYNAMIC HOOKS POINTS [Acherkan et al] are useful while *paving* as they allow a place to override the default paving, allowing for more flexibility when needed.

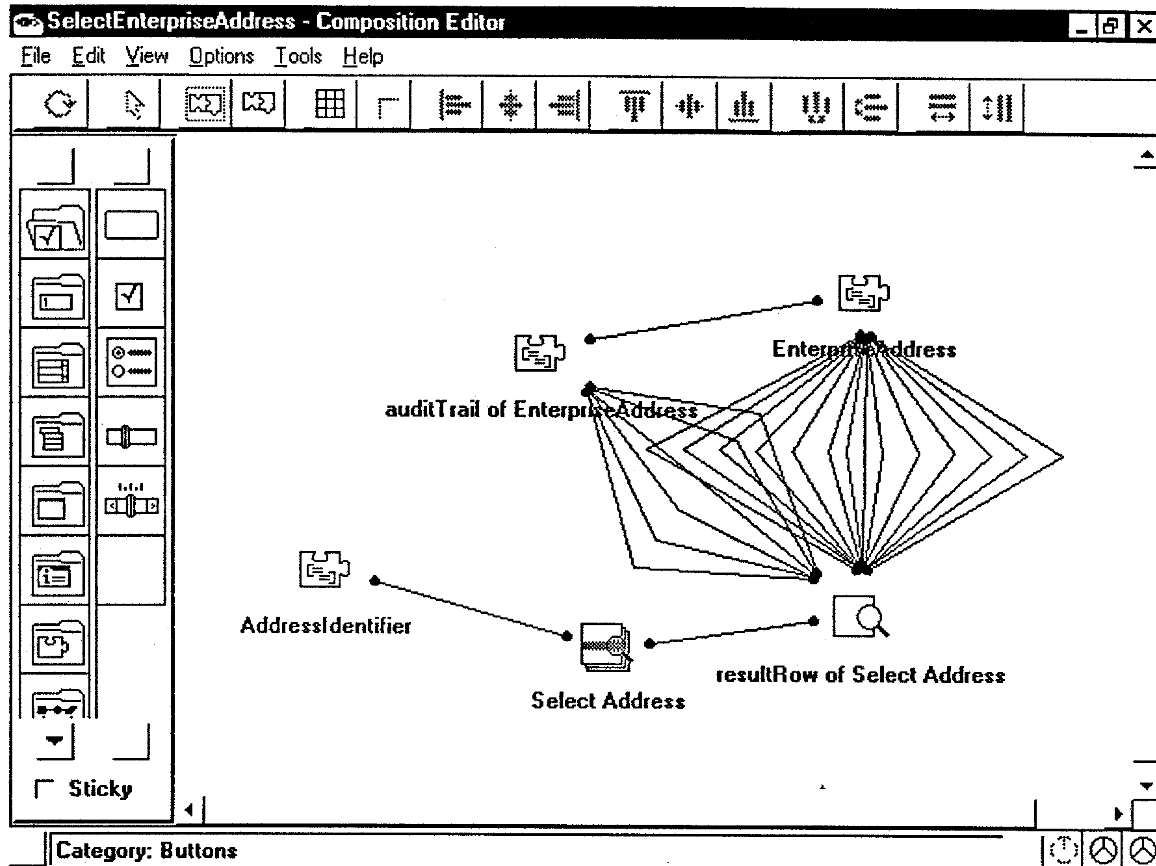

Figure 4 –A simple visual program example

2.9 Known Uses

There are many known uses, some outlined above. They include:

- GUI & persistence 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 The Refactory did this for the previously described Invoicing and Order Processing system as well as for defining medical rules for an Illinois Department of Public Health system.

3. PATTERN: WIPING YOUR FEET AT THE DOOR

ALIAS: ENCAPSULATE AND IGNORE / KEEPING THE INTERNALS CLEAN

3.1 Context

Some parts of your system are good, but there are pressures to change the API and functionality of existing good components to support other poorly designed components. These clean parts of the architecture are often core to the system, sometimes reusable, and need to change independently of other parts of the system. You want the good parts of your architecture to stay clean and not be tightly coupled to other parts of the system.

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 components that interact with it?

3.3 Forces

There are several forces:

- *Pressure to develop interfaces required by muddy code:* You'd like to keep the interfaces to functionality provided by components of your system simple and straightforward. However, you need to interact with other systems that may be poorly architected and don't conform to your architecture. Can you integrate these muddy systems without compromising your existing interfaces?
- *Ongoing Development:* As new functionality is added or existing functionality is modified, programmers routinely add code to integrate with external components and systems. Can they do so with minimal impacts to existing core components?
- *Limited ability to change existing, possibly muddy, components outside your control:* You need to integrate your software with external components. These components are not easily modified and may be poorly designed. How can you preserve the design integrity of your system's core components while integrating with existing components?
- *Need to accept a variety of requests from external systems:* Your system handles requests from several external systems. These systems often send requests and data in different formats. How can you architect your system to accommodate these variations?
- *Need to integrate existing inflexible systems with your components:* You'd like to define a common format for identical requests from external systems, but it is not possible to make them conform to a single interface specification. How can you support variations in requests and data formats without muddying up your implementation?

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 compromised. One way to do this is to implement an ADAPTER or BRIDGE that takes muddy requests and translates them into calls that preserve the "clean" interface. This cleansing and transformation can also be implementing using a PROXY or FAÇADE.

Figure 5 shows one possible implementation of wiping your feet at the door. In this solution a separate component performs some cleansing, transforming, and/or data validation before delegating the request to the "clean" components. This pre-processing allows you to wipe the feet in order to keep the "clean" component's processing consistent and uncompromised (note S1 to Sn can have varying APIs). 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, however, you will be able to deal with different variances in client code including possibly having a security proxy that validates and protects the internal component. You may even develop pluggable adapters that implement different rules depending on the client.

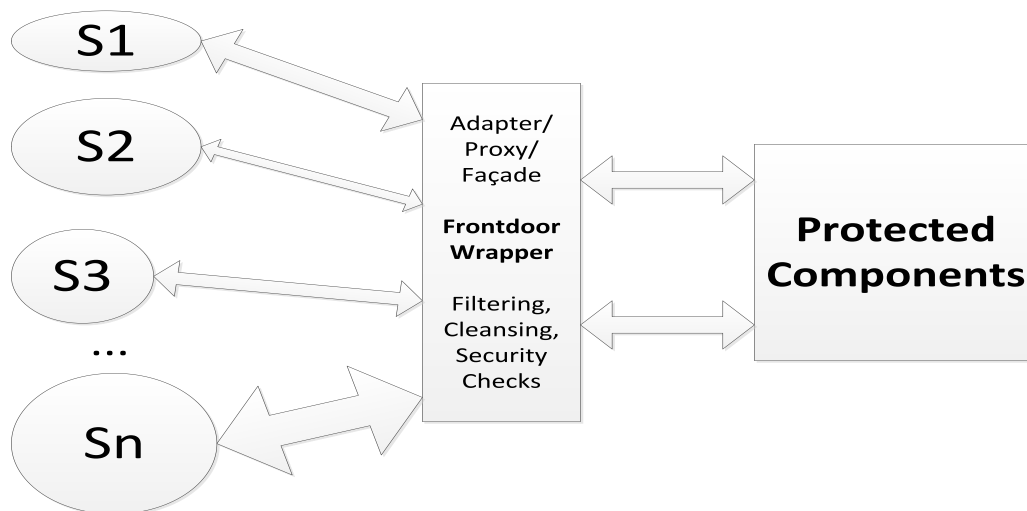

Figure 5 –Generic Filtering/Cleansing Wrapper Architecture View

Without this new API cleansing the data, you would have to directly couple your internal component to the external systems that your system interfaces to. This could muddy up the code and make it more difficult to support new variations. By separating out the cleansing and filtering functionality into separate, client specific components (the adapters or proxies), you allow the common internal component to remain clean and focused on its main responsibility. This also helps solidify the boundaries between different parts of the system ensuring clear roles and distinct responsibilities. These boundary areas are also where trust regions can be defined and enforced. Both input and outputs to and from the protected component can be filtered and transformed to provide the desired results.

3.5 Consequences

Advantages:

- Preserves the integrity of the clean component and its desired interface.
- Simplifies the implementation of the core "clean" component.
- Provides a place to link in alternatives including un-trusted code; thus allowing for validation and verification along with filtering and cleansing.
- Separates transformation/validation from code that implements component functionality.
- Ability to add one or more customized interfaces to support muddy code without breaking/changing existing code.
- Ability to validate requests without adding complexity to existing components.
- Allows components to be more loosely coupled.

Disadvantages:

- It may not be possible to completely contain the mud especially when new behaviors need to be added to the core component in order to support its muddy clients.
- Can decrease performance with translation layers between components.
- Increased complexity of an extra adaptation layer to interact with components. Mud can start to grow in the adapter but it can also be contained.
- Multiple interfaces, both clean and muddy, increase system complexity.

3.6 Examples

Figures 6 presents an architectural view of a system implemented by The Refactory that processed imported inventory data according to client-specific business rules. Some clients could import customer-specific inventory data to the system in various formats including flat files from main frames, CSV files, XM files, Excel files, etc. Each file format varied according to specific formatting rules defined by different client systems. The core system for importing and processing placed orders contained common code used regardless of file format being processed. In the example below, various formats of different customer files from different sites would first be cleansed, transformed, and validated according to client-specific rules. This pre-processing enabled the core PlaceOrder component to only handle canonical requests. This cleaning, transformation, and validation process is where we wiped our clients' feet at the front door and prevented client-specific code from compromising the integrity of our core processing code.

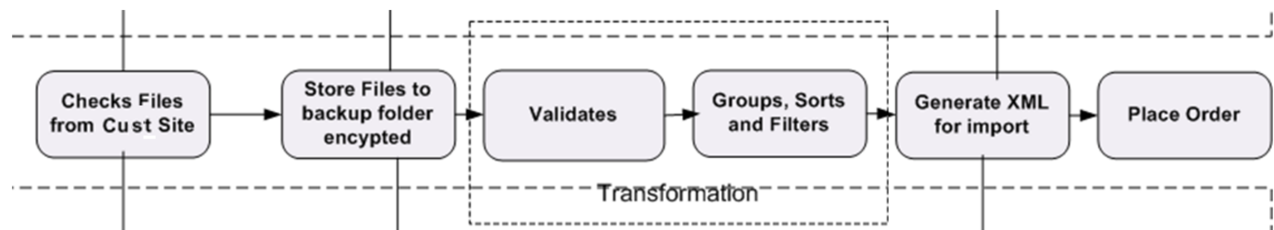

Figure 6–Filtering and Cleansing Sequence to keep Place Order Interface Clean

In addition to wrapping this core processing, we also put in DYNAMIC HOOKS POINTS at places where we could apply client-specific rules inside the core components without having to modify them. This was another important strategy we used to prevent "muddying up" the core. This allowed us to adapt to constantly changing formatting requirements.

Another example of Wiping Your Feet at the Door, shown in Figure 7, is one where Wirfs-Brock Associates worked with a telco client to integrate back office OSS applications including 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. External APIs were inconsistent, and often provided more functionality than was needed. 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.

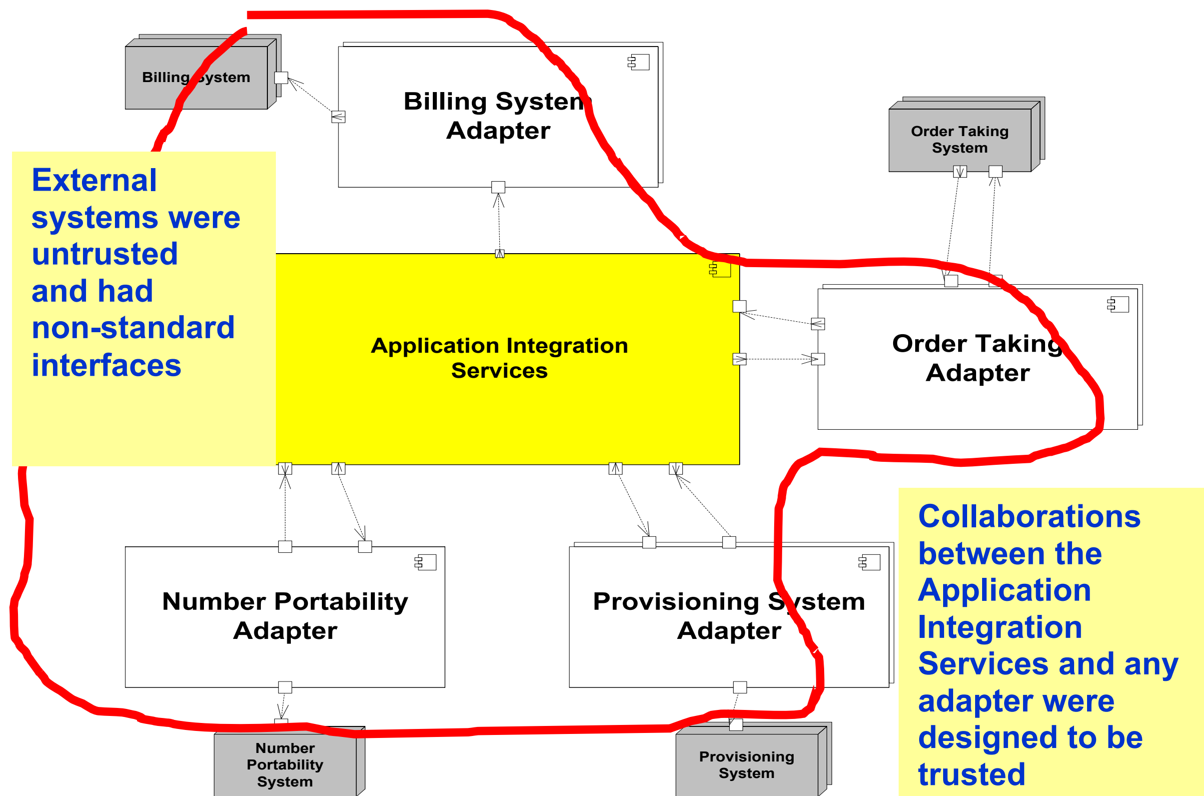

Figure 7 – Architecture of the OSS Integration System showing Trust Boundaries

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, FAÇADE, PROXY [Gamma et al] and FILTER [Buschmann et al] are possible ways to implement WIPING YOUR FEET AT THE DOOR.

INTERCEPTORS [Schmidt et al], which are dynamically invoked, can be used to filter and do pre-processing before invoking core components.

A MEDIATOR [Gamma et al] can be used to implement an Adaptor that has to communicate and adapt to multiple core components. The mediator provides the glue code to keep the core components clean.

SWEEPING UNDER THE RUG [Foote and Yoder] is an alternative pattern that helps protect other parts of the system by hiding a mess from other clean components or external systems.

3.8 Known Uses

The order processing systems by Refactory and Wirfs-Brock Associates mentioned in the above examples are some known uses the authors have implemented.

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. In Struts, much of the functionality is implemented via Interceptors. Features like double-submit guards, type conversion, object population, validation, file upload, and page preparation are all implemented using Interceptors. Each Interceptor is pluggable, so programmers can 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] in addition to 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

This paper presents two new patterns, PAVING OVER THE WAGON TRAIL and WIPING YOUR FEET AT THE DOOR. They capture proven practices for sustaining complex and often muddy systems. They do so by shoring up architectural boundaries, identifying and preserving core functionality, and providing easier ways accomplish repetitive programming tasks. A common misconception of BBoMs that we hope to dispel is that they are anti-patterns. Instead, we suggest that that many patterns in Big Ball of Mud pattern are often appropriate for simplifying complex systems, whether your goal is to clean up or contain mud or not. Our patterns complement several patterns described in the original Big Ball of Mud paper including SHEARING LAYERS and SWEEPING IT UNDER THE RUG.

There are many forces and good reasons that can lead to overly complex architectures. In fact, architects and top development teams are often doing exactly the right thing when they end up with some mud and unnecessary complexity 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 made in the original BBoM paper was that in spite of the best intentions, good decisions often lead to muddy architecture. Only in hindsight can you see what might have been a better, less muddy solution. Our contribution in this paper is to document patterns for improving architectures. Given the imperfect nature of evolving architectures for complex software systems, we hope to identify even more patterns that can help sustain system architectures.

5. ACKNOWLEDGEMENTS

We are grateful to Michael Stal for his valuable insight and input during the PLoP shepherding process. We’d also like to acknowledge Kirstin Heidler, Nicco Kunzmann, Francois Trudel, Marko Leppanen, Michael John, Tony Edgin, Jiwon Kim, Youngsu Son, Juan Reza, Eduardo Guerra and Ernst Oberortner, our writers’ workshop participants, whose thoughtful comments and questions and suggestions helped us clarify our patterns.

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. *Pattern Languages of Programs Design 4,* edited by Neil Harrison, Brian Foote, and Hans Rohnert., Addison-Wesley, 2000.

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.

Don Roberts and Ralph Johnson. "Patterns for Evolving Frameworks", *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.

Douglas Schmidt, Michael Stal, Hans Rohnert & Frank Buschmann. *Pattern-Oriented Software Architecture Vol.2: Patterns for Concurrent and Networked Objects*, Wiley & Sons, 2000.

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).